UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Marcelo T. Pereira

RIFFS: Um Sistema de Arquivos para Memórias Flash baseado em Árvores Reversas

Dissertação de Mestrado submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Orientador:
Antônio Augusto Medeiros Fröhlich

Florianópolis, Fev de 2004

RIFFS: Um Sistema de Arquivos para Memórias Flash baseado em Árvores Reversas

Marcelo T. Pereira

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação, área de concentração Sistemas Operacionais e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Raul Sidnei Waszlawick

Banca Examinadora

Antônio Augusto Medeiros Fröhlich

Marcelo Pasin

Rômulo Silva de Oliveira

Wolfgang Schröder-Preikschat

“A melhor forma de prever o futuro é criá-lo.”
(Peter Druker)

“Às minhas alianças afetivas,

à natureza,

ao futuro...”

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho apresenta uma nova estrutura de armazenamento de dados em memórias flash, chamada de *Reverse-Indirect Flash File Sistem* (RIFFS) ou **Um sistema de Arquivos para memórias Flash baseado em Àrvores Reversas** . As memórias flash possuem uma limitação na atualização de seus dados, e pensando em amenizar esta característica pensou-se em deixar todos os dados e meta-dados dentro do próprio arquivo. Isso seria impraticável com os sistemas existentes, porque não seria possível localizar um arquivo diretamente, a partir do `nodo raiz` da árvore. A maneira encontrada foi criar uma árvore reversa. Este esquema quebraria a navegabilidade do sistema, e então uma árvore direta precisa ser construída na memória RAM. É mostrado neste trabalho o gerenciamento de uma árvore reversa para contornar as limitações das memórias flash. Dentro deste esquema é possível evitar excessivas atualizações e operações de escrita, aumentando assim a vida útil da flash.

Keywords: sistemas operacionais, sistemas embutidos, memória flash, sistema de arquivos.

Abstract

This project presents a new technique for flash storage management called a *Reverse-Indirect Flash File System* (RIFFS). However, flash memories have a drawback: its data cannot be updated-in-place. To solve this limitation, all the data and meta-data is left inside of the file itself. This would be impracticable with the current systems, because it would not be possible to locate a file in a directory tree, as is usually done. The solution was to construct a reverse-tree. This schema would break the navigability of the system, and then a direct tree need to be constructed in RAM memory. This work shows the reverse-tree management schema to solve the limitations of flash memories. This solution helped to minimizate extreme updates and write operations, increasing flash life-time.

Keywords: operating systems, embedded systems, flash memory, file systems.

Capítulo 1

Introdução

Já não é de hoje que o homem moderno se acostumou com aparelhos eletrônicos em seu cotidiano, provendo funções de processamento ou armazento de dados. Alguns trazem vantagens no desempenho de seu trabalho atuando como ferramentas, enquanto outros trazem conforto, comodidade, etc. Estes aparelhos estão presentes nas mais diversas áreas, desde a mais simples como um relógio despertador, até as mais complexas como por exemplo o sistema de foco de uma filmadora digital ou o sistema de navegação de um carro [5]. Em muitos casos, para o funcionamento destes dispositivos, existe a necessidade de um software de configuração e controle, e a todo este conjunto damos o nome de **sistemas embutidos**.

Os sistemas embutidos são construídos com componentes eletrônicos em geral, como por exemplo circuitos integrados, portas lógicas, circuitos impressos, microprocessadores, componentes programáveis, etc; e comumente controlados por softwares específicos. Normalmente quando nos referimos à este tipo de circuito, lembramos de computadores pessoais (PCs) e de seus componentes de processamento (CPU), esquecendo dos vários equipamentos a nossa volta que também se utilizam deles. De acordo com Tennenhouse [16] apenas 2% dos 8 bilhões de processadores fabricados em 2000 foram aproveitados para estações de trabalho (PCs), sendo que a grande maioria teve seu fim em sistemas embutidos.

O sucesso dos sistemas embutidos não se deu apenas à utilização da

tecnologia de componentes eletrônicos, mas também ao uso de software específico para cada sistema. Estes programas podem tanto exercer apenas algumas funções dentro do sistema como executar funções mais complexas de gerenciamento de processos, alocação de recursos, etc.

Independentemente da complexidade e do tamanho, este software específico precisa ser armazenado em algum tipo de mídia não-volátil, e a mais comumente usada é a memória apenas de leitura (Read-Only Memory - ROM), e suas variantes: EPROM[1] e EEPROM[2]. Este último tipo de memória foi eleita pela sua baixa latência de leitura, alta resistência e pequeno tamanho, em comparação à outras mídias. Quando é necessário atualizar algum dado, memórias deste tipo devem ser ser apagadas por inteiro e depois reescritas. A fim de melhorar a rotina de atualização, foi agregado mais um membro à esta família de memórias, chamado de **memória flash**. Esta não precisa ser apagada por inteiro, mas em blocos chamados de unidades de apagamento ou apenas **setores**. Com isso sua atualização acaba sendo mais rápida e conseqüentemente seu consumo de energia acaba sendo menor.

Com uma alta densidade, um peso leve, um pequeno tempo de latência, um baixo consumo de potência, e por fim uma vantagem na atualização de dados, as memórias flash se tornaram um meio atrativo de armazenamento de dados. Infelizmente por causa do seu preço, hoje em dia ela não é usada como armazenamento principal em computadores pessoais (ou outros sistemas de grande porte), mas suas vantagens de armazenamento em sistemas embutidos é clara.

Como dito anteriormente, existe o grupo de sistemas embutidos que está ficando cada vez mais complexo, como é o caso do telefone celular, por exemplo. Esta linha de dispositivos precisa se preocupar, entre outras coisas, com o armazenamento: dos dados de configuração, dos dados do usuário, de módulos do próprio sistema (como por exemplo uma nova versão de máquina virtual Java), etc. Por causa do tamanho das aplicações, houve uma necessidade do aumento das memórias flash, e da implementação de um sistema de arquivos. A manipulação de dados dentro destas memórias possui

[1] EPROM - Eraseable ROM

[2] EEPROM - Electrically EPROM

suas peculiaridades, e por este motivo, a construção de um sistema de arquivos em uma memória flash torna-se um desafio à engenheiros de software.

O principal desafio deste trabalho consiste no desenvolvimento de uma estrutura de armazenamento diferente dos sistemas de arquivos para memórias flash convencionais. Neste cenário, cada arquivo agrupa todas informações pertencentes a ele próprio, e este trabalho nomeou esta estrutura como **contexto de arquivo**. Conforme é mostrado nos próximos capítulos, com a adoção desta estrutura é possível uma diminuição na atualização dos dados de controle em relação a outros sistemas de arquivos. Para conseguir satisfazer este requisito, foi preciso a concepção de uma **árvore reversa** com a ligação **indireta** entre seus nodos, surgindo assim o nome do projeto: Reverse-Indirect Flash File System - RIFFS.

O próximo capítulo mostra as desvantagens da memória flash, e alguns algoritmos para contorná-los. O terceiro capítulo é um estudo sobre as teorias clássicas de sistemas de arquivos, e seus algoritmos. O quarto capítulo mostra como são feitos os sistemas de arquivos para memórias flash. O quinto capítulo descreve o sistema de arquivos proposto (RIFFS), sua arquitetura e projeto. No sexto capítulo é mostrado como foi feita a implementação do primeiro protótipo. Por último, é mostrado a conclusão do trabalho.

Capítulo 2

Memórias Flash

Este capítulo descreve uma visão geral da tecnologia de memórias flash, trazendo principalmente o estado-da-arte neste campo e servindo como base para as próximas seções neste texto.

A memória flash é um tipo de memória não volátil, mas com seu funcionamento bem distinto. Ela trabalha como a união das características de leitura e escrita das memórias de acesso aleatório (conhecidas como memórias RAM) com as características de armazenamento das unidades de disco magnético. O armazenamento dos dados dentro dessas memórias é dado em células, como nas RAMs dinâmicas (conhecidas como memórias DRAM), mas em geral são usadas como um disco magnético pelo fato da persistência de dados quando a energia é desligada. Por causa da sua alta velocidade, sua grande resistência contra impactos, seu tamanho reduzido e seu baixo consumo de potência, as memórias flash tornaram-se um meio ideal para armazenamento em várias aplicações embutidas como câmeras digitais, telefones celulares, impressoras, roteadores, tocadores de MP3, etc [5].

2.1 Conceitos Gerais

Memórias flash são similares às tão conhecidas memórias EEPROM, e a principal diferença entre elas está no fato de que as memórias flash são apagadas

somente em blocos, e não por inteiro como é feito nas EEPROMs, possibilitando assim, criar um sistema para gerenciar dados dentro dela, uma vez que não é preciso perder toda sua informação a cada apagamento. Pelo fato do apagamento ser baseado em setores, os circuitos das memórias flash acabam sendo mais simplificados, permitindo assim uma maior densidade em relação a uma memória EEPROM equivalente.

Existem atualmente vários tipos de tecnologias de memórias flash, as quais podemos citar: *NOR, DINOR, T-Poly, AND, NAND*; e cada uma dessas tecnologias requer um gerenciamento (funções de leitura, escrita e apagamento) específico [15]. Estas tecnologias permitem às memórias flash, reterem dados sem uma fonte de energia por períodos longos como 20 anos, por exemplo. No entanto, essa mesma tecnologia é responsável por um dos grandes problemas das memórias flash: a **limitação no número de apagamentos**, por causa do desgaste das células de armazenamento. Com isso, os fabricantes precisam fixar o número de apagamentos que garanta a integridade dos dados (por exemplo: 100.000 apagamentos), e atualmente, este valor é razoável para a maioria das aplicações embutidas.

Pelo fato das memórias flash serem apagadas por setor, estes requerem uma atenção especial quanto ao seu tamanho. Os fabricantes, além de produzirem memórias com setores de diferentes tamanhos, ainda produzem *chips* com diferentes tecnologias, que também são conhecidos no mercado como memórias flash híbridas. Como característica, ainda podemos citar a proteção por hardware em alguns setores da flash, para evitar o apagamento indevido.

2.2 Operações

Existem três operações básicas que podem ser realizadas em uma flash: **leitura**, **escrita** e **apagamento**. O tempo de leitura e escrita de uma flash é normalmente equivalente às mesmas operações em uma DRAM, mas o tempo da operação de apagamento é bem maior (chegando perto da casa dos segundos). Nos próximos parágrafos é descrito em detalhes as operações possíveis de uma flash.

Leitura: A leitura de uma memória flash é bastante parecida com as leituras em memórias voláteis convencionais. Para ler um dado, basta disponibilizar o endereço desejado no barramento de endereços da memória, e capturar o dado do barramento de dados. Isso torna o acesso dos dados em memórias flash mais rápido que os meios magnéticos.

Com o intuito de reduzir os acessos e aumentar a quantidade de informação lida, alguns fabricantes implementam em suas memórias flash, sofisticados métodos de acesso, como: **buffer de páginas**, e **leitura seqüencial**. No primeiro método o *chip* contém uma memória volátil interna que armazena temporariamente os dados, permitindo a leitura de uma página inteira. O segundo método é conhecido como rajada (burst), onde é preciso informar apenas o primeiro endereço de uma leitura seqüencial de dados.

Escrita: O formato da escrita dos dados em uma flash é um pouco diferente daquele que estamos acostumados a pensar. Uma flash é dita apagada quando todos seus bits possuem o nível lógico 1, e dentro desta filosofia, para escrevermos algum dado em uma flash, é necessário trocar alguns bits para 0. Nesse modo, podemos resumir a operação de escrita como sendo: escrever zeros. É importante lembrar que o contrário, transformar zeros em uns, só é possível no setor inteiro. Assim como na leitura, ainda temos os respectivos métodos sofisticados para escrita, como: **buffer de páginas** e **escrita seqüencial**.

Apagamento: Como dito anteriormente, o apagamento se dá em um setor inteiro da flash. Ele é um comando distindo para a memória, assim como a leitura ou a escrita. O resultado da sua operação pode ser exemplificado como escrever todos os bits do setor para o valor lógico 1. Pelo fato do apagamento ser a operação mais demorada de uma flash, existe um método de **espera** para melhorar o desempenho do sistema. Com este método é possível parar a operação de apagamento para realizar outro tipo de acesso ao dispositivo.

2.3 Tecnologias

Desde a invenção das memórias flash, fabricantes têm procurado alternativas para aumentar o desempenho e a capacidade desses dispositivos. Como essas memórias ganharam novos mercados, tecnologias foram incorporadas para fazer das flashes um produto mais competitivo no mercado de armazenamento. O avanço destas memórias e suas tecnologias mais importantes estão citados abaixo:

Tamanho de setor variável: alguns modelos possuem setores de tamanhos variados que permitem uma melhor manipulação do sistema de arquivos. Esses setores de tamanho variados são necessários quando se deseja bloquear dados na flash. Normalmente os setores de tamanho variado em uma flash estão no início ou no fim do seu espaço de endereçamento.

Paralelismo de operações: as flashes podem ser formadas por diferentes bancos de armazenamento que operam em paralelo, e assim, operações podem ocorrer simultaneamente na mesma flash, desde que estejam sendo feitas em diferentes bancos.

Interface padronizada: Common Flash Interface (CFI) é um conjunto de operações adotado por um grupo de fabricantes com a intenção de padronizar o acesso às informações das memórias flash. Por exemplo, uma memória flash possui informações como seu número de série, número do fabricante, número de setores, etc. O CFI foi o padrão adotado para a requisição dessas informações, para que as aplicações não se preocupem em conhecer os detalhes dos dispositivos e das versões de cada fabricante.

Tecnologia de vários níveis: essa tecnologia se refere à capacidade de armazenar dois bits de informação em apenas uma célula. Normalmente uma célula consegue armazenar apenas um bit de informação, e com esta tecnologia, o tamanho do *chip* está sendo reduzido pela metade.

Segurança: alguns modelos possuem registradores de segurança que indicam qual setor na flash pode ser protegido. Este termo segurança não está relacionado com

criptografia, mas sim com o bloqueio físico de um setor na flash.

Setores híbridos: alguns modelos de memórias flash podem ter setores de diferentes tecnologias. Estes dispositivos são chamadas de **híbridos** e encontrados em fabricantes que disponibilizam memórias com setor de *boot*. Normalmente este setor de *boot* é uma tecnologia diferente dos outros setores porque possui uma maior compatibilidade com as memórias RAMs e EEPROMs, e são acessadas diretamente por um dispositivo de processamento no seu início de execução. O preço que se paga por esta compatibilidade é o seu desempenho, ou seja, este setor possui um atraso na leitura dos dados, em comparação aos setores de outras tecnologias.

2.4 Estudos de Casos

Esta seção mostra alguns fabricantes de memórias flash bem como sua tecnologia disponível no mercado.

2.4.1 AMD

AMD disponibiliza vários modelos de memórias flash compatíveis com a interface CFI. As voltagens de seus dispositivos variam entre 1.8V e 5.0V. A densidade máxima hoje em dia é de 256 Mb com a presença, ou não, de um setor dedicado para *boot*. Suas memórias operam em temperaturas entre a faixa comercial (de 0 a 145^o C) até a faixa super-estendida (de -55 a 145^o C). Os modelos disponíveis por este fabricante possuem atualmente as tecnologias chamadas de *MirrorBit* e *Dual Operation*, explicadas a seguir.

Dual Operation: Pela incorporação de uma memória *SRAM* no *chip*, a AMD fez um modelo que pode funcionar com várias operações simultâneas. Todas operações são executadas na memória *SRAM* e depois transferidas para flash. A aplicação não tem essa visibilidade de operação, e usa o componente como se fosse uma flash normal.

MirrorBit Technology: Esta é a tecnologia de vários níveis, que diferencia esse tipo de flash das demais, pela implementação de uma célula que armazena dois bits de informação. Dessa maneira o *chip* tem metade do tamanho se comparado com outro de mesma capacidade.

2.4.2 Intel

Os produtos da Intel também possuem suporte à interface CFI. Os componentes operam a uma faixa de voltagem entre 1.8V e 5.V, e alguns modelos necessitam uma voltagem maior (aprox. 12V) para realizar operações especiais como bloquear um setor, etc. Os tipos de dispositivos fabricados podem ter as seguintes tecnologias: paralelismo (vários bancos), tamanho de setores variados, setores híbridos e células de vários níveis. A Intel tem ainda duas tecnologias exclusivas que minimizam o tempo de escrita na flash, chamados de *Enhanced Factory Programing* (EFP) e *Buffered Enhanced Factory Programing* (BEFP), explicadas a seguir.

EFP e BEFP: Essas duas tecnologias são usadas para diminuir o tempo de escrita na flash, usadas normalmente em sistemas embutidos que não possuem interação com o mundo externo após entrarem em funcionamento. Para isso é preciso pré-gravar a flash antes de colocá-la no sistema definitivo. Apesar do nome diferente, essas duas tecnologias realizam a mesma tarefa, mas a diferença entre elas está no fato que o BEFP possui células de vários níveis.

2.4.3 ATMEL

As memórias da ATMEL funcionam com uma voltagem variando entre 2.7V e 5.0V. A capacidade de armazenamento das memórias desta companhia variam desde 32 Mbits até 512 Mbits. As freqüências de leitura podem chegar até 100MHz. Em alguns modelos, é possível definir o tamanho do barramento (variando entre 16 e 32 bits) em tempo de execução. As tecnologias mais usadas são: paralelismo e setor de *boot*. Este fabricante possui três tipos principais de memórias flash no mercado, conhecidas como

Fast Programming Time, *Serial Flash* e *Data Flash*, explicados a seguir.

Fast Programming Time: É uma tecnologia para flash que possui um segundo estado de voltagem para escrita de dados. Esse segundo estado é aproximadamente 12V e isso diminui o tempo de escrita das memórias deste modelo.

Serial Flash: este modelo implementa o seu protocolo de comunicação via Interface de Periféricos Seriais (SPI). Esta interface faz com que a flash possa ser usada como uma memória substituta das EEPROM SPI, sem nenhuma mudança no *layout* da placa (caso a pinagem seja compatível). Este *chip* opera em 20MHz, e sua densidade varia desde 512 Kbits até 4 Mbits.

Data Flash: Flash compatível com a interface SPI. Pode ter alguns setores híbridos, e o seu acesso pode ser serial ou paralelo, dependendo do número de bancos do dispositivo.

2.4.4 MICRON

Os produtos da MICRON são compatíveis com a interface CFI. As voltagens dos dispositivos variam entre 2.7V e 5.0V. O tamanho do barramento pode ser escolhido em tempo de operação, variando entre 8 ou 16 bits. As temperaturas variam da comercial à estendida. Alguns modelos possuem a tecnologia de *boot-sector* e multi-bancos. O principal modelo é chamado de *Sync Flash* que possui uma interface *SDRAM* com o mundo exterior, fazendo o software enxergá-la como uma memória do tipo RAM.

Sync Flash: Neste modelo, uma interface SDRAM é implementada. Com um alto desempenho de leitura (similar à uma RAM equivalente), esse tipo de flash tornou-se uma escolha competitiva para aplicações que necessitam executar código, ao invés de simplesmente armazenar dados. Os planos para o futuro desta tecnologia é substituir as atuais memórias RAM para persistência de programas.

Capítulo 3

Sistema de Arquivos

O Sistema de Arquivos serve para dar suporte ao armazenamento de arquivos de vários tipos, como textos, desenhos, programas executáveis, etc. Entre outras tarefas, o sistema de arquivos deve prover para o usuário uma interface simples e fácil de usar, na manipulação de seus dados. Ele é responsável por implementar em software um recurso que não existe no hardware. O hardware oferece simplesmente um grande conjunto de bytes contíguos, e a tarefa principal do sistema de arquivos é implementar a abstração de arquivo em cima do dispositivo de armazenamento. Este capítulo trata dos conceitos relacionados aos dispositivos de armazenamento e ao gerenciamento de arquivos e diretórios pelo sistema de arquivos, descritos a seguir.

3.1 Dispositivo de Armazenamento

O dispositivo de armazenamento de um sistema de arquivos pode ser qualquer mídia (também chamada de memória secundária), a qual provê um meio de armazenamento em massa, e a **persistência de dados**[1]. Algumas tecnologias de dispositivos de dados persistentes, utilizam o acesso a dados através de blocos físicos, como é o caso de discos óticos como CD ou DVD, discos magnéticos rígidos ou flexíveis, etc. Um **bloco físico** delimita a unidade de leitura ou escrita no dispositivo. Em cada operação

[1]Persistência também pode ser entendida como a retenção de dados previamente armazenados, sem fonte de alimentação.

deste tipo um bloco inteiro é copiado da memória para o dispositivo (escrita) ou do dispositivo para a memória (leitura). O tamanho de cada bloco físico pode variar de acordo com cada fabricante, e no intuito de padronizar o acesso a dados, os sistemas de arquivos usam uma estrutura chamada de **bloco lógico**, obviamente de tamanho maior ou igual ao dos blocos físicos dos dispositivos.

Outra característica importante no acesso a dados é preocupação com o **gerenciamento de blocos livres**. O sistema de arquivos precisa saber quais blocos estão ocupados e quais estão livres, ou desocupados, quando da inserção de novas informações, evitando a escrita de dados em cima de informações já inseridas anteriormente. A seguir são mostrados os conceitos de: blocos lógicos e seu gerenciamento.

3.1.1 Blocos Lógicos

O conceito de Blocos Lógicos surgiu da necessidade em homogeneizar as operações em diferentes dispositivos. Desta forma as camadas de mais alto nível dos sistemas podem trabalhar com blocos lógicos de qualquer tamanho, fixo ou variado, sem se preocupar com as peculiaridades específicas de cada dispositivo, implementadas pelas camadas de acesso ao hardware. Um fator relevante para um sistema é o **tamanho do bloco lógico** utilizado. Este tamanho pode ser fixo ou variado, conforme mostrado a seguir:

Blocos de tamanho fixo: Dentro deste cenário, existe um fator muito importante na escolha do tamanho do bloco, que é a granularidade do disco. Por um lado, um dispositivo muito grande com blocos pequenos pode ser de difícil gestão, enquanto que um dispositivo muito pequeno com blocos grandes pode apresentar uma fragmentação interna indesejada.

Blocos de tamanho variado: Um sistema com blocos de tamanho variado apresenta uma maior flexibilidade do sistema em tempo de execução. Por outro lado, o código que implementa este tipo de característica, precisa ter um cuidado maior no controle de seus blocos. Existe a preocupação, a cada operação, do tamanho do bloco, o que não ocorre com blocos de tamanho fixo.

3.1.2 Gerenciamento de Blocos Livres

O Gerenciamento de Blocos Livres é uma das tarefas de um sistema de arquivos que adota a padronização de blocos lógicos, mostrados anteriormente. Esta função é de extrema importância para o sistema, pois um simples erro pode sobrescrever uma área utilizada, chegando até a invalidar um arquivo inteiro.

Este gerenciamento é extremamente dependente do tipo do bloco lógico de dados adotado pelo sistema (fixo ou variado), e também do tipo de mapeamento dos blocos (descrito a seguir). No entanto, temos basicamente duas técnicas para o gerenciamento de blocos livres: **mapa de bits** e **lista encadeada**, conforme mostrado a seguir.

Mapa de Bits: Dentro do dispositivo é reservado um espaço onde será inserido este mapa. Ele consiste de uma seqüência de bits, onde a posição do bit indica o número do bloco que ele representa, e seu valor indica o estado do bloco que pode ser livre ou ocupado. A vantagem deste método é a sua simplicidade de implementação e a facilidade na detecção de blocos contíguos para alocação. Sua desvantagem vem da dificuldade em gerenciar grandes mapas, uma vez que não podem ser carregados por inteiro na memória principal.

Lista Encadeada: Consiste em manter uma lista encadeada contendo todos os blocos livres do disco. Para alocar um bloco, retira-se o primeiro da lista e para liberar adiciona-o na lista. Esta lista é grande no caso de um dispositivo vazio e normalmente ela é mantida na própria mídia. Conforme a ocupação do dispositivo aumenta, esta lista diminui até sua extinção, provendo seu espaço inicial para o usuário (o que não ocorre no conceito anterior). Ela é bastante eficiente em operações corriqueiras de alocação e liberação, mas escritas aleatórias pode gerar blocos seqüenciais completamente dispersos.

3.2 Gerenciamento de Arquivos

A manipulação de dados nos dispositivos pode conter um conjunto de atividades difíceis e indesejadas pelos usuários, como o cálculo da sua localização, cont-

role de alocação, etc. A fim de tornar estas atividades transparentes, a funcionalidade do sistema de arquivos é passada aos usuários através do conceito de arquivo.

Arquivo é um conjunto de dados armazenados em um dispositivo. Cada arquivo contém dados do usuário que possuem algum significado para ele ou para o sistema. Normalmente os arquivos possuem um nome dado pelo usuário para que este seja identificado entre os demais arquivos dentro do sistema. Além do nome, cada arquivo pode possuir uma série de outros atributos que são úteis tanto para o usuário quanto para o sistema, e entre os mais usuais podemos citar: tipo do conteúdo, tamanho, data e hora de criação, Data e hora de alteração, permissões de acesso, etc.

Os arquivos são vistos pelo sistema através de uma estrutura chamada **descritor de arquivo** (*file descriptor*). O descritor é um registro no qual são mantidas as informações a respeito do arquivo. Essas informações incluem: os seus atributos, além de outros dados que não são visíveis aos usuários, mas imprescindíveis para que o sistema implemente as operações sobre arquivos. Um exemplo destes dados é o número lógico atribuído a cada *file descriptor*, também chamado de identificador e conhecido por **id**.

3.2.1 Operações

O sistema de arquivos deve prover um conjunto de operações para que o usuário manipule seus arquivos. A partir das operações básicas, muitas outras podem ser implementadas e exportadas como facilidades do sistema. Um exemplo é a operação de cópia de arquivo, a qual é implementada com as operações de leitura e escrita. Diferentes sistemas de arquivos, implementam diferentes funções básicas, mas podemos citar como as mais usuais:

- `Criação(create):` Cria um arquivo sem dados, e um descritor lhe é associado. Caso não existam descritores disponíveis no dispositivo de armazenamento, a solicitação de criação é negada.
- `Remoção(remove):` Operação que libera os recursos associados ao arquivo.
- `Abertura(open):` A fim de acessar dados contido em um arquivo, um processo

deve antes abrí-lo. Nesta operação, o descritor de arquivo é trazido para as tabelas internas do sistema, na memória principal, para o rápido acesso.

- `Fechamento(close):` Esta operação indica ao sistema de arquivos, que o processo não precisará mais acessar os dados do arquivo, e a tabela interna do sistema é atualizada.

- `Posicionamento(seek):` Operação que atribui um valor ao ponteiro de dados do arquivo. Este ponteiro é utilizado nas funções de leitura e escrita.

- `Leitura(read):` Operação responsável por ler dados de um arquivo. O ponteiro de dados indica onde começa a leitura (o posicionamento do ponteiro é feito através da função `seek`). Nesta função é preciso indicar ainda a quantidade de dados a serem lidos, e a posição de memória que os dados serão copiados.

- `Escrita(write):` Função parecida com a `leitura`, só que nesta operação os dados são escritos. O ponteiro de dados indica onde começa a escrita. É preciso indicar a quantidade de dados a serem escritos e a posição da memória que contém os dados. Esta operação também pode ser chamada de **expansão**(`append`), caso o ponteiro de dados esteja na última posição do arquivo.

- `Leitura Atributos(stat):` Função responsável pela visualização dos atributos de um arquivo.

- `Escrita Atributos(chmod):` Esta operação possui a responsabilidade de escrever atributos em um arquivo.

3.2.2 Gerenciamento dos Blocos de Arquivos

O sistema precisa se preocupar com algumas características que todo gerenciamento de arquivos genérico deve possuir. Podemos citar, entre outras, como as mais comuns:

- Criação de arquivos com grandes dados;

- Possibilidade de acesso seqüencial a arquivos;
- Possibilidade de acesso direto a arquivos;
- Possibilidade de expansão de arquivos;
- Possibilidade de alteração do conteúdo de arquivos.

Estas e outras características são possíveis de acordo com o **mapeamento** dos dados do arquivo para os blocos lógicos, e conseqüentemente da arquitetura do descritor de arquivo. Este mapeamento está normalmente dentro do descritor de arquivo. É através dele que é possível encontrar os dados de cada arquivo.

O mapeamento pode ser realizado de três formas básicas (e mais uma série de formas mistas):

- **alocação contígua:** É a forma mais simples para alocar espaço em um dispositivo. Cada arquivo ocupa uma seqüência contígua de blocos. No descritor de arquivo é preciso manter o endereço do bloco lógico no qual o arquivo se inicia e o tamanho. As grandes vantagens deste método são a simplicidade do mapeamento e o pouco gasto de espaço para manter a informação dos dados do arquivo. O tempo do método `seek` acaba sendo rápido, pois é implementado com um cálculo simples de deslocamento. A desvantagem aparece quando é preciso aumentar o tamanho do arquivo. Caso não exista blocos livres contíguos suficientes após o fim do arquivo, todo seu conteúdo precisa ser copiado para outra região do dispositivo que acomode todos os seus dados e mais a quantidade de blocos que se deseja expandir.
- **alocação encadeada:** Este tipo de gerenciamento de blocos serve para contornar a limitação da alocação anterior. Neste cenário, cada bloco contém no seu interior o endereço do próximo bloco e assim por diante. Deste modo o descritor de arquivo continua o mesmo, armazenando apenas o bloco inicial e o tamanho do arquivo, mas uma parte de cada bloco físico é gasto para manter um endereço para o próximo bloco. A vantagem deste método está em permitir que qualquer bloco livre possa ser alocado a qualquer arquivo, sem uma alocação contígua no dispositivo. Como

desvantagem, este método não permite o acesso direto a seus dados, fazendo com que a função `seek` seja lenta, gastando muito tempo com *I/O* para ler a lista de blocos encadeados.

- **alocação indexada:** Dentro deste tipo de mapeamento, o descritor de arquivo é implementado como uma **tabela de índices** (diferentemente das duas implementações anteriores). Neste esquema, cada entrada da tabela contém o endereço de um dos blocos que formam o arquivo. Assim é possível contornar as duas desvantagens anteriores. De um lado, ele não necessita da alocação contígua de blocos, e de outro, ele não precisa ler os blocos na operação de `seek`. Uma questão importante a ser tratada neste cenário é o tamanho da tabela de índices dentro do descritor de arquivo. Este tamanho tem que ser avaliado de tal forma que, possam ser construídos arquivos grandes e pequenos sem consumir muito espaço. Uma técnica muito usada neste tipo de alocação é o uso de níveis de indireção na indexação, presente nos sistemas Unix [3], como é o caso do *Extended File System II* (EXT2) [11]. Desta maneira, a tabela de índices pode ser pequena e acomodar uma grande quantidade de dados, através de índices **diretos** e **indiretos**.

3.3 Gerenciamento de Diretórios

O termo **Diretório** pode ser entendido como sendo um conjunto de arquivos ou conjunto de referências a arquivos. Eles são úteis para organizar os arquivos no sistema, ou seja, são eles que nos permitem organizar os arquivos em grupos, facilitando sua localização. Isto é particularmente útil quando um usuário deseja visualmente localizar um arquivo. Ao agrupá-los em diretórios, as listas de arquivos podem ser pequenas, facilitando a visualização do arquivo.

As referências a arquivos são guardadas dentro do diretório, em forma de tabela, que por sua vez pode conter qualquer informação desejada pelo engenheiro de software. Cada linha desta tabela referencia um arquivo do sistema, e a esta referência é dado o nome de **entrada de diretório** ou então **entrada de arquivo** [14].

O simples fato de como esta tabela é disposta, e quais são suas informações, ditam como a estrutura de diretório pode ser formada. Existem diversas formas de estruturar os diretórios de um sistema, entre as mais básicas, podemos citar:

- **diretório linear:** Também conhecido como *flat*, é a forma mais simples de estruturar o diretório de um sistema. Neste caso o sistema possui somente um diretório, e este corresponde a uma lista de todos os arquivos existentes no dispositivo. Como desvantagem não é possível separar os diferentes arquivos, impossibilitando o usuário de organizar seus arquivos em lugares separados, ou agrupá-los conforme sua necessidade. Neste caso, todos arquivos, tanto do usuário quanto do sistema, ficam em um mesmo lugar.

- **diretório em dois níveis:** Para dar mais flexibilidade ao primeiro sistema, esta implementação disponibiliza dois níveis de diretórios. Desta maneira, o sistema possui uma lista de diretórios, e cada diretório possui uma lista de arquivos. Assim, é possível que o usuário agrupe seus arquivos em diretórios, mas não é possível a criação do terceiro nível de diretórios.

- **diretório em árvore:** É possível extender o conceito de diretórios de tal forma que os usuários também possam criar livremente os seus próprios diretórios e subdiretórios. Desta forma os diretórios são implementados dentro do sistema como uma estrutura do tipo *arquivos*. Cada arquivo precisa possuir um **tipo** que o classificará como **arquivo de usuário** ou **arquivo de sistema**. O resultado é um sistema organizado em forma de árvore, e cada usuário tem a possibilidade de organizar seus arquivos da maneira mais conveniente.

- **diretório em grafo:** Dentro deste esquema, os diretórios continuam sendo implementados como arquivos. Dentro de uma entrada de diretório é encontrado o nome do arquivo, alguns atributos e uma referência (normalmente um número) do arquivo. Assim, pode-se ter um mesmo arquivo com dois nomes diferentes e em lugares diferentes.

3.3.1 Operações

As operações básicas mais comuns que podem ser realizadas sobre os diretórios são descritas a seguir:

- `Criação(mkdir):` Cria um diretório vazio.
- `Remoção(rmdir):` Remove um diretório vazio.
- `Inserção de ítem(link):` Insere um ítem em um diretório. Seus parâmetros mais comuns são o nome do arquivo e alguns de seus atributos.
- `Remoção de ítem(unlink):` Remove um ítem de um diretório.

As outras operações como leitura e escrita de atributos são responsabilidade da implementação de arquivos (sessão 3.2.1), e por isso não é mostrado nesta seção.

Capítulo 4

Sistemas de Arquivos para Memórias Flash

Atualmente as memórias flash estão sendo usadas como um padrão de armazenamento de dados de sistemas embutidos em geral. No entanto, tornar um simples *chip* de memória flash em um sistema complexo de armazenamento de dados não é uma tarefa simples. Pensando em aproveitar as vantagens da flash, e tentando contornar suas limitações, pesquisadores tiveram que desenvolver e reciclar conceitos para construir sistemas de arquivos para essas memórias, tornando eficiente a manipulação de dados. Esses sistemas de arquivos são normalmente implementados em dois modos: alguns desenvolvidos por inteiro [20] enquanto que outros são construídos dentro de uma camada de software de acesso ao dispositivo (também conhecido como *driver*), mantendo assim uma compatibilidade com as camadas superiores dos sistemas de arquivos existentes [6].

Este capítulo mostra o uso das memórias flash em sistemas embutidos através do ponto de vista de sistemas operacionais, incluindo *drivers* e sistemas de arquivos. É importante lembrar que este trabalho não possui a intenção de esgotar o assunto de sistemas de arquivos para memórias flash, mas sim fazer uma análise abrangente sobre os sistemas existentes e seus algoritmos.

4.1 Conceitos Gerais

Apesar das várias vantagens da memória flash, ela apresenta algumas limitações, que podem ser visualizados como desafios para os engenheiros de software: nenhum dado podem ser **reescrito**, e ao invés disso ele tem que ser apagado antes. Para isso o setor tem que ser apagado por inteiro, e ainda há que se tomar cuidado com o **número de apagamentos** que é limitado. Para contornar este inconveniente, vários algoritmos e conceitos foram propostos desde o começo do mercado dessas memórias, e esta seção trata especificamente destes algoritmos.

Sistemas de arquivos tradicionais possuem a propriedade de atualização provido pela natureza dos seus dispositivos (como os discos magnéticos, por exemplo). Isso faz com que os dados em um setor do disco possam ser atualizados, quantas vezes for necessário, mas isso não acontece com as memórias flash. O esquema de atualização de dados nessas memórias pode ser conseguido de duas formas: **apagamento-e-escrita** e **remapeamento** de dados, mostrados a seguir.

4.1.1 Apagamento-e-escrita

Esta estratégia é representada pela figura 4.1, onde é mostrado o estado inicial de uma flash com um dado de nome *data_1* sendo atualizado, mostrado em 4.1(a). Para realizar esta operação é preciso: 4.1(b) copiar os dados válidos de todo o setor para um setor temporário[1]; 4.1(c) apagar o setor; 4.1(d) copiar o novo dado e os dados do setor temporário e 4.1(e) apagar o setor temporário. Pelo fato desta estratégia sempre gastar o apagamento de dois setores a cada atualização, esta técnica não é implementada pelos sistemas de gerenciamento destas memórias. Isso sem contar na quantidade de processamento e tempo gasto para sua realização.

[1] É importante lembrar que na figura 4.1(b) a cópia dos dados válidos precisa ser necessariamente para um dos setores da memória flash, pois se forem copiados na memória RAM do sistema, e ocorrer alguma interrupção na alimentação elétrica, os dados seriam perdidos.

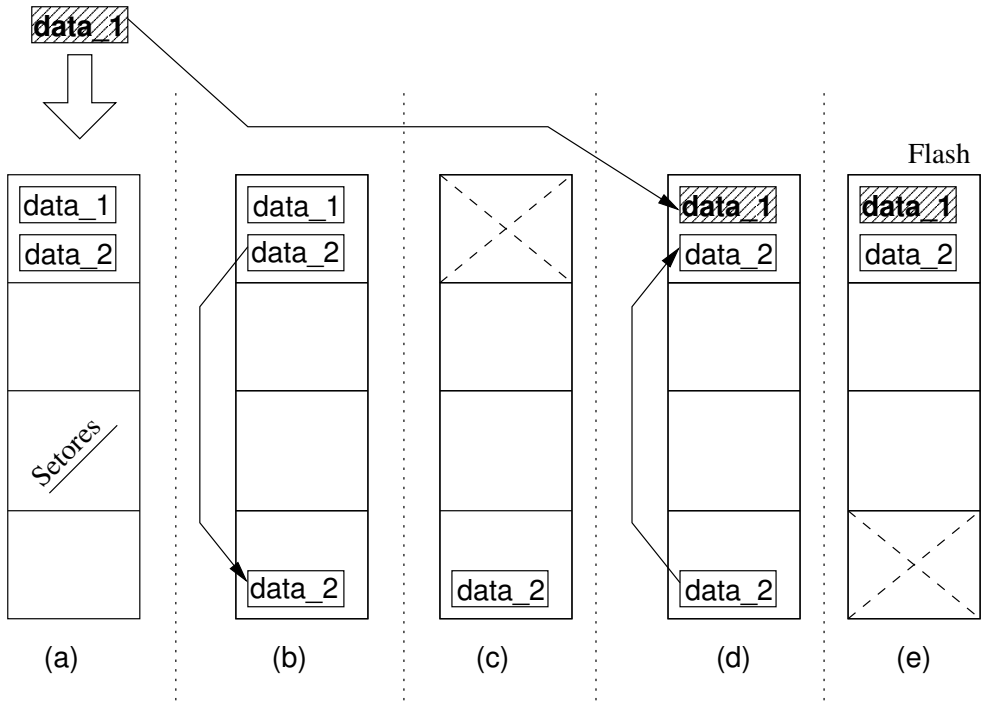

Figura 4.1: Atualização de dados na Flash.

4.1.2 Remapeamento

O esquema de **remapeamento**, mostrado na figura 4.2 consiste em gravar a atualização dos dados em lugares diferentes dos originais, necessitando de uma tabela (normalmente na memória RAM) para fazer a tradução dos dados válidos. Através da figura 4.2 é dado um exemplo de como é feita a atualização de um dado. Suponha um setor da flash com dois dados distintos, chamados de `data_1` e `data_2`. Estes dois dados são apontados por seus respectivos ponteiros na memória RAM. No exemplo da figura 4.2(a) o `data_1` precisa ser atualizado. Para executar esta tarefa de atualização, todo o dado precisa ser escrito em uma parte vazia da flash, conforme mostrado em 4.2(b). Após realizada esta tarefa o ponteiro do `data_1` na memória RAM precisa ser atualizado para a nova posição do dado, mostrado em 4.2(c). Como desvantagem, este método de remapeamento causa uma fragmentação nos setores por causa dos dados inválidos, necessitando assim de um **procedimento de limpeza** para apagá-los posteriormente.

Convém reforçar que a estratégia de apagamento e escrita para atualização

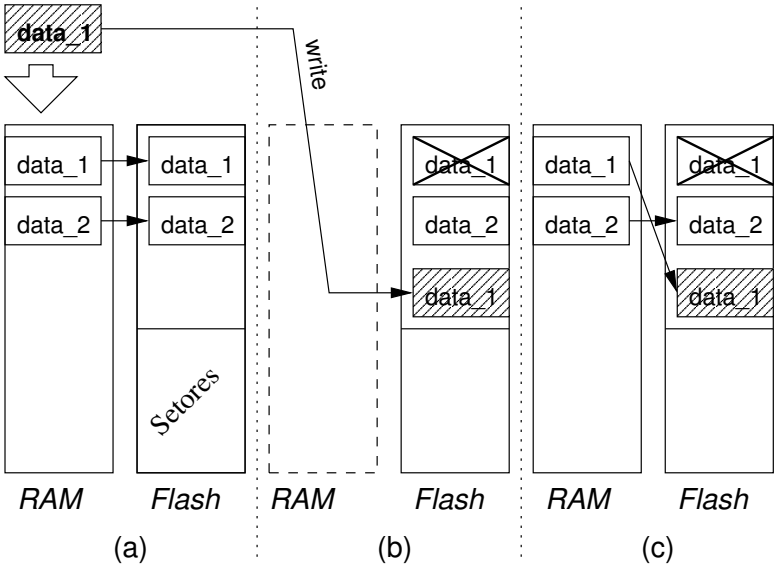

Figura 4.2: Atualização de dados na Flash.

de dados em uma flash é desaconselhada para um sistema de arquivos, porque diminui a vida útil dessas memórias, uma vez que o número de apagamento dos setores é limitado. Por esse motivo, os algoritmos de atualização de dados em memórias flash são sempre implementados através do conceito de remapeamento.

Limpeza de Setor: A estratégia de apagar um setor, reorganizando seus dados válidos em outro lugar, é chamada de limpeza de setor, e seu procedimento é conhecido como **coletor de lixo** (*garbage collect*). Para realizar esta função com eficiência, muitos estudos sobre **políticas de limpeza** foram implementados. De acordo com Chiang [2], a escolha dessas políticas de limpeza possui um grande impacto no desempenho do sistema, podendo reduzir a eficiência de uma aplicação em até 50%.

As políticas de limpeza levam em consideração três aspectos fundamentais, como: **seleção** do segmento a ser limpo, a **reorganização** dos dados, e o **início da rotina** de limpeza. O primeiro conceito leva em conta quais e quantos segmentos devem ser limpos. Com isso o coletor de lixo tem a oportunidade de trabalhar com uma maior variedade de arquivos, podendo realizar com mais eficiência a estratégia de reorganização dos dados. A reorganização se preocupa em ‘como agrupar os diferentes dados (por exem-

plo, dados do mesmo arquivo ou do mesmo diretório). Já o terceiro conceito se preocupa quando será o início da rotina, que por sua vez pode ser realizada de três modos: por tempo determinado, por porcentagem de utilização da flash, ou por um processo de baixa prioridade no sistema que está sempre fazendo esse serviço.

Em cima desses métodos, pesquisadores desenvolveram conceitos para ter um melhor desempenho no gerenciamento de dados dentro de uma flash. A maioria dos estudos tem se voltado para os problemas do coletor de lixo, com o objetivo de **reduzir o número de apagamentos** e o **número de páginas copiadas**.

Estratégias do Coletor de Lixo: Existem atualmente, vários esquemas para realizar com eficiência a limpeza de um setor. O primeiro método proposto foi através de uma política gananciosa (*greedy*), que por sua vez recicla o setor que possui o maior número de dados inválidos. Estudos como o de Kawaguchi [6] mostram a ineficiência desse método. Esses pesquisadores propuseram então uma outra estratégia para limpeza do setor, chamada de: custo-benefício (*cost-benefit*). Dentro deste esquema, é atribuído um valor[2] para cada dado escrito no setor, e o coletor de lixo executa seu método de limpeza com base nesses valores.

Chiang [9] , melhorou o desempenho do coletor de lixo tirando proveito da localidade dos acessos e classificando os dados como quente ou frios (*hot-cold*). Dentro deste esquema, os dados são agrupados de acordo com a sua taxa de atualização, sendo assim, os dados mais antigos, tendem em ficar no mesmo setor. Douglis [7] fez um estudo estatístico do coletor de lixo. Ele afirma que a eficiência de um sistema diminui significativamente quando a utilização da memória flash é alta. Como exemplo, ele explica que quando a utilização da flash for aumentada de 40% para 95%, o tempo de resposta das operações de escrita pode cair até 30%, e o tempo de vida da flash pode ser reduzido a um terço.

[2]Este valor pode ser entendido como a data de escrita dos dados

4.2 Device Drivers

Algumas memórias flash possuem um encapsulamento[3], exportando assim uma visão de disco magnético para o sistema operacional. Isto permite acoplar memórias flash em sistemas despreparados para esta finalidade. Este não é o nosso caso, uma vez que esse estudo se volta em para um eficiente gerenciamento da memória flash, via software. Com isso esta seção mostra a primeira camada de software (presente em uma memória volátil) sobre essas memórias, conhecidos como gerenciador de dispositivo, gerente de dispositivo ou piloto de dispositivo (*device driver*).

4.2.1 Estudo de Casos

Os *device drivers* para memórias flash podem ser implementados de dois modos: emulando um disco magnético ou exportando rotinas básicas de manipulação do dispositivo. No primeiro caso, o piloto de dispositivo gerencia a flash por inteiro (implementando *wear levelling, garbage collector, etc*), exportantdo para as camadas de aplicativos de um sistema operacional, blocos de tamanho pequeno (aproximadamente 512 bytes) como a emulação de um disco. No segundo caso, o gerente de dispositivo exporta para o sistema operacional rotinas básicas de manipulação do dispositivo como leitura, escrita e apagamento, deixando para as camadas superiores de software a responsabilidade de realizarem o gerenciamento de dados dentro da memória flash (*wear levelling, garbage collector, etc*). Como exemplo do primeiro caso, podemos citar *Flash Translation Layer* (FTL), e como exemplo do segundo caso podemos citar *Memory Technologi Driver* (MTD), explicados a seguir.

Flash Translation Layer (FTL): FTL é uma camada de software de gerenciamento para memórias flash que exporta a visão de um disco magnético para o sistema operacional. Na montagem do volume, toda a memória é lida para então ser contruído na memória RAM uma mapa do sistema. Este mapa é entendido como um conjunto de ponteiros para posições específicas na flash que serão exportadas como blocos físicos de

[3]Encapsulamento pode ser entendido como um hardware adicional.

um disco magnético. Pela teoria, a FTL habilita qualquer sistema de arquivos para disco magnético a ser instalado sobre uma flash, mas normalmente quem faz muito uso desta camada são os sistemas para Windows como o VFAT por exemplo.

O compromisso principal da FTL é tornar compatível as implementações anteriores de sistemas de arquivos, para que continuem acessando a memória como um dispositivo de bloco. Este foco não é aceitável por este trabalho, pela redução de desempenho gasto com processamento de ponteiros de blocos e pelo consumo desnecessário de espaço no sistema. Estes pontos podem ser ressaltados de acordo com o exemplo a seguir.

Conforme dito anteriormente, para cada pequeno setor emulado, é necessário um ponteiro dentro de uma tabela de mapeamento que, em muitos casos, é armazenada no próprio dispositivo, mas mantida na RAM. Suponha como exemplo uma memória flash de tamanho igual a 128MB. Supor que cada ponteiro de bloco seja igual a 4 bytes, e o tamanho do bloco igual a 512 bytes. Dividindo o tamanho da flash pelo tamanho do bloco, teríamos 256k blocos. Multiplicando o total de blocos pelo tamanho do ponteiro, teríamos 512kB sendo usados como ponteiros em uma tabela na flash, que também pode estar replicada na RAM. A cada atualização esta tabela precisa ser alterada, e com o passar do tempo, o setor a qual ela pertence será reciclado pelo coletor de lixo, necessitando de uma lógica adicional para ser reescrita em outro setor. Note que não está sendo levado em conta a tabela que o sistema de arquivos cria, que também possui seu tamanho e gerenciamento.

Memory Technology Driver (MTD): A camada MTD [19] é uma especificação de software recente dentro de projetos do sistema operacional Linux, principalmente para a área de sistemas embutidos. O projeto MTD define uma interface genérica para acesso a dispositivos de memória, em particular, dispositivos flash. Além de exportar uma interface de **pequenos blocos** para uma possível emulação de disco, o driver MTD ainda exporta uma interface de acesso a **simples caracteres**, que permite aos sistemas de arquivos possuir uma visão da flash como uma memória linear de dados[4].

[4]Esta visão de simples caracteres é usada pelo *Journalling Flash File System* (JFFS) mostrado na sessão 4.3.

O foco do projeto MTD é definir uma interface padrão entre um dispositivo e um sistema operacional. Neste sentido, alguns gerenciadores de dispositivo dentro do projeto são implementados apenas com funções básicas de acesso ao *hardware*, sem se preocupar com algoritmos de gerenciamento de memórias flash como um coletor de lixo por exemplo. O sistema MTD pode ser dividido em dois modos de operação: `modo usuário` (*user mode*) e `modo dispositivo` (*device mode*). O primeiro se caracteriza por um conjunto de módulos que exporta uma interface de alto-nível para as camadas de aplicação, já o segundo é um conjunto de funções simples, de acesso ao dispositivo como: leitura, escrita e apagamento de dados. Acima destas duas visões é possível encontrar sistemas de arquivos complexos, com é o caso do *Journalling Flash File System* (JFFS), ou simplesmente gerentes de dispositivo, como é o caso da camada *Flash Translation Layer* (FTL).

A idéia de fornecer uma interface padrão, com funções básicas, para acesso ao dispositivo (como é o caso do `modo usuário` do MTD), foi capturada e adaptada para o projeto do RIFFS. Neste trabalho é implementado uma interface de acesso ao dispositivo mostrado no capítulo 6.

4.3 Sistemas de Arquivos

Programas de aplicação podem possuir a funcionalidade de armazenar e buscar qualquer dado de uma memória flash através de serviços do *device driver*. No entanto, pode-se tornar inadequado o fato de, em sistemas embutidos, controlar diretamente os dados armazenados, principalmente em dispositivos que requerem uma atenção especial, como é o caso das flashs. Se diferentes aplicações precisam manipular dados aleatoriamente, ou se a manipulação de dados for muito intensa, então a instalação de um sistema de arquivos é a melhor solução.

Mesmo os sistemas de arquivos sendo tão importantes no gerenciamento de dados, poucos foram propostos para memórias flash. A maioria deles, se baseiam no fato de ter uma camada que emula um disco magnético, permitindo que sistemas de arquivos não específicos para memórias flash sejam instalados sobre estas memórias.

Felizmente, um sistema de arquivos específico possui mais vantagens, em questão de desempenho, em relação à outros sistemas classificados como **genéricos**, uma vez que os primeiros possuem a oportunidade de manipular diretamente as limitações impostas pela tecnologia. Esta seção traz um estudo de casos sobre alguns sistemas de arquivos existentes atualmente.

4.3.1 Estudo de Casos

Os sistemas de arquivos para memórias flash são normalmente implementados de dois modos distintos: alguns desenvolvidos por inteiro, enquanto que outros fazem apenas a parte de gerenciamento de dados. O primeiro, implementa todos os algoritmos de gerenciamento de dados na flash e ainda exporta suas funcionalidades para as aplicações. Esses sistemas são chamados de específicos, e podemos citar como exemplo: o *Journalling Flash File System* [17], e o *Embedded File System* [12]. Como exemplo do segundo modo de implementação, temos o *True Flash File System* [10].

Journalling Flash File System (JFFS): O *Journalling Flash File System* [17] implementa um sistema de arquivos específico para memórias flash, levando em conta sistemas embutidos. A versão número um, conhecida como JFFS1, foi implementada como um sistema de arquivos com estrutura em registro, conservando o funcionamento e algumas estruturas descritas no *Log-Structured File System* (LFS) [13]. Por causa de suas desvantagens com o coletor de lixo, e pensando em acrescentar algumas características, Woodhouse deu início a construção da segunda versão, que acabou ficando conhecida como JFFS2 [18].

A segunda versão do JFFS melhorou as desvantagens do JFFS1, e ainda partiu para a portabilidade do seu código para todas as plataformas que possuem o Linux como sistema operacional, dando prioridade aos sistemas embutidos. A eficiência do coletor de lixo foi a segunda grande vantagem do JFFS2 sobre a primeira versão. Ele conseguiu um melhor desempenho mudando o esquema da reciclagem de uma simples lista circular para um sistema de gerenciamento de blocos ponderado. Segundo esse método,

o algoritmo de limpeza faz decisões de qual setor será reciclado, o que não acontecia na primeira versão. Ainda para melhorar a versão, foi adicionado compressão de dados que pode ser usado caso o usuário configure essa opção.

As estruturas do JFFS2 que ficam na memória RAM são construídas na inicialização do sistema. Este início envolve uma operação de quatro passos: **leitura** da memória flash e alocação de todos os nodos na memória RAM, **apagamento** das estruturas que contém dados inválidos, **apagamento** das estruturas que não possuem referência e **apagamento** dos dados temporários. Feito isso, o sistema de arquivos começa a sua operação.

Pela natureza de seu contexto o JFFS2 foi desenvolvido com controle de usuários, integrado ao sistema operacional que o suporta. Ele utiliza blocos lógicos de tamanho fixo, seu gerenciamento é do tipo alocação encadeada, e seu gerenciamento de diretórios é do tipo grafo.

Quando é citado blocos lógicos de tamanho fixo em sistemas embutidos, os engenheiros de software de tais sistemas tem que se confrontar com uma grande diversificação dos arquivos, diferente do estudo feito para sistemas do tipo Unix, onde a maior taxa de arquivos é de tamanho pequeno [8]. Temos, cada vez mais, diferentes tipos de arquivos e tamanhos dentro de um sistema embutido. Hoje, podemos ter arquivos de alguns bytes como pequenos módulos do sistema operacional e até arquivos de alguns megabytes como um filme dentro de uma filmadora digital. Como exemplo de armazenamento e gerenciamento de ponteiros para blocos lógicos (a nível de sistema), pode ser tomado um filme de 100MB. Levando-se em conta que cada bloco lógico seja de tamanho fixo igual a 1kB, e cada ponteiro para bloco seja de 4 bytes, seria utilizado 400kB apenas com a referência de um arquivo. No projeto RIFFS, tenta-se contornar o uso desnecessário de vários ponteiros para blocos de tamanho fixo adotando uma estrutura de blocos de tamanho variável, conforme tratado no capítulo 5.

Nos sistemas de arquivos que possuem a árvore de diretórios em forma de grafo, como é o caso de vários sistemas de arquivos do sistema operacional Linux, cada diretório possui uma lista de referências para descritores de arquivo. Cada referência é chamada de entrada de diretório. Cada entrada de diretório possui algumas informações

do arquivo, como por exemplo o nome e a localização do descritor de arquivo no dispositivo de armazenamento. Suponha um diretório em uma flash com com 100 arquivos, por exemplo. Se o usuário por algum motivo, mudar o nome de todos os arquivos, ficariam cem referências inválidas no início dos dados do diretório, e cem referências válidas no final. Esta seria a forma normal de atualização de dados dentro de uma flash, de acordo com o capítulo 4. Quando o coletor de lixo escolher o setor no qual os dados do diretório estão inseridos, será preciso um processamento adicional para copiar apenas as referências válidas para outro setor. No projeto RIFFS tenta-se reformular o conceito de entrada de diretório utilizando uma estrutura chamada de `contexto de arquivo`, visto em detalhes no capítulo 5. Com isso, a responsabilidade de referências ao arquivo é repassada para esta estrutura.

Quando se trata de controle de usuários em sistemas embutidos, refere-se a sistemas e aparelhos, normalmente utilizados por apenas uma pessoa. Quando acontece este caso, as informações de controle de usuário tornam-se desnecessárias para o sistema em questão. Isto também é válido para outras informações de um sistema mais complexo, como é o caso do JFFS2. Através da literatura, não foi possível encontrar meios para eliminar o controle de usuário do sistema no JFFS2 por exemplo, acrescentando assim um *overhead* em aplicações que não o necessitam. Muitas vezes, em sistemas embutidos, não existe a necessidade do controle de usuário, e como exemplo podemos citar uma filmadora digital que não utilize este recurso. No projeto RIFFS, não existe este controle de usuário, eliminando processamento adicional em aplicações que não o necessitem.

Embedded File System (Efsys): O Efsys da *QNX Software Systems* [12] combina as funcionalidades de um sistema de arquivos junto com as de um *device driver*. Por esse fato, existem várias versões do sistema, cada uma desenvolvida para um tipo de fabricante de memórias flash.

O software suporta dois tipos de partição: **partição simples** e **sistema de arquivos**. A primeira pode ser entendida como qualquer setor na flash que não necessite dos algoritmos de gerenciamento de dados. Como exemplo, podemos ter a imagem de

um componente do QNX, que não precisará de atualização. O segundo tipo de partição, é a mais comum, onde se encontram as estruturas do sistema, seus dados de controle, etc. O formato de armazenamento de informações é proprietário, e os diretórios e arquivos são organizados como uma lista encadeada de **nodos**. Um nodo pode ser entendido como uma faixa contígua de bytes em um dispositivo (pode ser uma flash, um disco, etc), e um arquivo pode ser formado por múltiplos nodos.

Quando a flash é formatada, alguns dados de controle são escritos no setor, mas um deles fica reservado para ser usado pelo coletor de lixo como armazenamento temporário na reciclagem de dados. Uma característica interessante do sistema de arquivos é a sua descompactação transparente dentro da função de leitura. O mesmo não ocorre na função de escrita, onde o usuário tem que explicitamente chamar a função de compactação.

Ao traçar um paralelo com o projeto RIFFS, encontramos os mesmos pontos ressaltados no JFFS2, exceto o controle de usuários, que o Efsys não implementa.

True Flash File System (TrueFFS): A empresa *M-Systems* implementou seu sistema de arquivos TrueFFS [10], baseado na camada padrão FTL, também patenteada por eles. Ele exporta a memória flash para o sistema operacional como um disco magnético. Por sua vez, não foi necessário desenvolver algoritmos de gerenciamento dessas memórias, porque o FTL provê essas funções de uma maneira transparente.

É necessário que exista um sistema de arquivos entre o sistema operacional e o TrueFFS. A Figura 4.3 mostra um exemplo das camadas envolvidas nesse sistema. O TrueFFS encapsula o módulo FTL e então exporta serviços de um disco magnético, além de realizar funções específicas de acoplamento com o sistema operacional.

Este sistema de arquivos possui as mesmas limitações impostas pela camada FTL. O desperdício de espaço e processamento para blocos de tamanho fixo na camada de acesso ao dispositivo é o mesmo. Em cima disto, os usuários do TrueFFS tem que se confromtar, com pontos levantados anteriormente no JFFS2 como blocos de tamanho fixo, e árvore de diretório em forma de grafo.

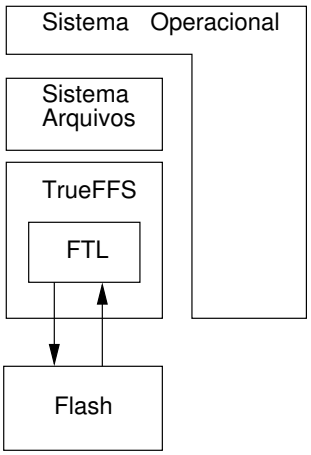

Figura 4.3: Camada TrueFFS dentro do Sistema Operacional.

Capítulo 5

Projeto do Sistema RIFFS

Através da literatura, percebeu-se que a maioria da pesquisa sobre sistemas de arquivos para memórias flash aconteceu com o intuito de aumentar o desempenho do sistema aprimorando os conceitos vistos anteriormente, como por exemplo o coletor de lixo. A proposta deste trabalho é mostrar uma nova estrutura de armazenamento de dados em memórias de difícil atualização, especialmente as memórias flash. Com isso espera-se: uma melhoria no desempenho dos coletores de lixo existentes, uma economia na atualização das estruturas e no processamento, e conseqüentemente um aumento na vida útil do dispositivo.

5.1 Objetivos

O principal objetivo deste projeto é evitar a complexidade das estruturas clássicas de sistemas de arquivos, e a atualização de dados. Para conseguir este objetivo, foi preciso uma nova arquitetura no gerenciamento de diretórios dentro das estruturas criadas na flash, chamado de árvore reversa, e o resgate do conceito de blocos lógicos de tamanho variado no gerenciamento de arquivos. Resumidamente, as características necessárias a este projeto estão listadas a seguir:

- **Simplicidade das estruturas:** No início do projeto, duas das principais metas eram: tornar as estruturas físicas armazenadas na flash o mais simples possível, para

economizar em espaço e evitar as atualizações, e simplificar o gerenciamento das entradas de diretório, ao longo de sua vida. Com o começo do projeto, percebeu-se que não só o espaço estava sendo economizado, como também a vida útil dos setores.

- **Arquivos possuem todas informações:** Este requisito surgiu da necessidade em eliminar as entradas de diretório das estruturas de armazenamento, para que não ficassem resíduos de um arquivo apagado dentro de um diretório. Desta maneira, as informações da entrada de diretório foram agrupadas com o descritor de arquivo, e a esta união, deu-se o nome de **contexto de arquivo**. Convém lembrar que esta estrutura, contexto de arquivo, é encontrada armazenada na memória flash, e serve para dar suporte para a construção do sistema na memória principal.

- **Navegabilidade do sistema:** Neste projeto, de acordo com suas características, um diretório não possui uma lista de referências à descritores de arquivos. Desta forma, não seria possível navegar na árvore de diretórios do sistema de arquivos. A solução adotada foi acrescentar ao contexto de arquivo uma referência para o diretório ao qual ele pertence, também chamado de diretório pai. Pelo fato do diretório ser implementado como um arquivo dentro do sistema, ele também possui um contexto, que por sua vez aponta para seu pai, e assim sucessivamente. Assim a árvore armazenada no dispositivo acaba sendo reversa. Como é impossível navegar em todos os sentidos em uma árvore assim construída, é necessária a criação de uma árvore de diretórios em memória principal, onde os filhos têm referência dos pais e os pais a dos filhos.

- **Blocos Lógicos:** Por causa da grande variação de tamanho dos arquivos em sistemas embutidos atuais, fica difícil prever qual o tamanho ideal da estrutura de dados, também chamada de bloco lógico. Foi pensando desta maneira que este projeto adotou uma estrutura de blocos de tamanho variável para o armazenamento de dados. Como gerenciamento de blocos de arquivos, o mapeamento adotado foi do tipo indexado.

- **Fragmentação:** Não existe fragmentação externa, neste projeto. Pelo fato do tamanho dos blocos ser variado, qualquer espaço pode ser alocado como um bloco lógico de dados.

5.2 Modelo Arquitetural

Para realizar todas as características descritas anteriormente, a arquitetura do sistema de arquivos pode ser mostrada através dos modelos de **gerenciamento de arquivos**,**gerenciamento de diretórios**, e **gerenciamento do dispositivo de armazenamento**, mostrados a seguir:

5.2.1 Sistema de Arquivos

Dentro deste projeto, o conceito de arquivo é definido como um conjunto de dados. Estes dados podem ser tanto de controle, como de usuário. De acordo com os requisitos deste projeto, foi necessário manter todas informações a respeito do arquivo dentro dele próprio. Para conseguir esta característica, o projeto RIFFS criou uma estrutura especial chamada de **contexto de arquivo**. A seguir é mostrado o funcionamento do contexto, e como é a estrutura interna de um arquivo. Cada arquivo possui um tipo, que os classifica como arquivo de usuário e diretório.

5.2.1.1 Gerenciamento de Arquivos:

Dentro do gerenciamento de arquivos encontramos basicamente três estruturas, que formam um arquivo: o contexto de arquivo, o mapa de blocos lógicos de dados pertencentes ao arquivo, e o seu tipo.

O contexto é responsável por agregar informações de controle e atributos, como por exemplo o nome do arquivo. O segundo atributo guarda todos blocos de dados pertencentes ao arquivo (caso existam), e seus respectivos tamanhos. O atributo tipo classifica o arquivo perante o sistema como: arquivo de usuário e arquivo de sistema, utilizado na implementação do diretório. Na figura 5.1(b) é apresentado um exemplo

de como um arquivo é visto quando carregado na memória RAM. Já na figura 5.1(c) é possível visualizar os blocos do arquivo espalhados pela memória flash. Em 5.1(a), é mostrado a visão do arquivo na forma de um conjunto de dados. Nesta figura, o círculo representa um arquivo de nome `file1.txt` que possui os blocos de dados identificados no exemplo por `f_1`, `f_2`, e `f_3`.

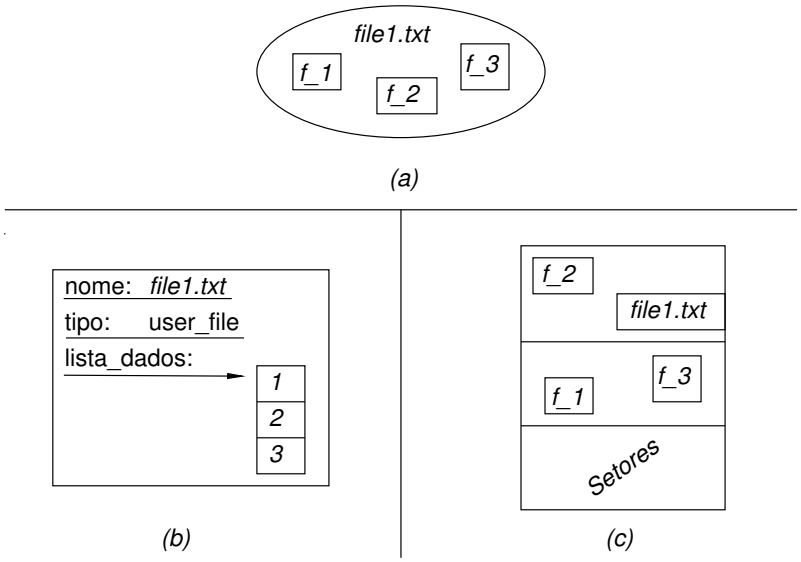

Figura 5.1: (a) Visão Lógica. (b) Visão da RAM. (c) Visão da Flash

Contexto de Arquivo: Todas informações de controle pertinentes ao arquivo estão dentro de seu contexto. O contexto é implementado como um bloco dentro da flash, e por este motivo, ele possui os mesmos atributos de um bloco lógico, descrito em 5.2.2. Ele possui em seus dados: uma referência para o contexto pai, e o nome do arquivo. A referência para o contexto pai é utilizado no gerenciamento de diretórios, e explicado a seguir. O nome do arquivo é uma sequência de caracteres e guarda o nome escolhido pelo usuário.

Organização dos blocos de arquivo: Cada bloco lógico de um arquivo possui uma versão que identifica as várias partes de um arquivo. Assim, ordenando a lista de blocos lógicos em uma forma crescente, temos os dados do arquivo organizados (mais detalhes sobre o campo versão de cada bloco lógico ver seção 5.2.2). Isso foi necessário para

permitir que os blocos do arquivo sejam reescritos de uma forma aleatória em qualquer parte do dispositivo, garantindo sua reconstituição na montagem do sistema. A figura 5.1(a) mostra dois tipos de blocos do arquivo `file1.txt`: blocos de usuário, e um bloco do tipo contexto. Como blocos de dados do tipo usuário, temos: `f_1`, `f_2`, e `f_3`, e o bloco do tipo contexto está representado pelo nome do arquivo: `file1.txt`.

5.2.2 Dispositivo Armazenamento

Uma memória flash foi usada como dispositivo de armazenamento. Este dispositivo não possui o conceito de blocos físicos, uma vez que manipula bytes em qualquer posição da memória. Isto é uma vantagem pois seus dados são acessados de uma forma aleatória, sempre com um mesmo tempo, pré-definido pelo fabricante, o que não ocorre nos discos por causa de sua natureza.

Arquitetura Interna: O setor é uma unidade muito importante no projeto de um sistema de arquivos para memórias flash, e por isso ele precisa ser analisado com cautela. A arquitetura física do dispositivo foi focado dentro do setor, e não dentro da flash. Neste trabalho ele possui uma estrutura mostrada através da figura 5.2. Nela é possível visualizar três estruturas básicas: estrutura de dados (presente na área de dados), estrutura de controle (presente na área de controle), e o cabeçalho (presente no início de cada setor).

A primeira estrutura pode ser entendida como os próprios dados do arquivo, e representam os blocos lógicos de tamanho variado. Eles são gravados no sentido do início para o fim do setor, e representado na figura 5.2 como a parte superior do desenho. Estas estruturas recebem o nome de `raw data`. Como previsto, o tamanho desses dados pode variar tanto quando se deseje, mas desde que não ultrapasse o valor máximo do setor. Caso isso ocorra, o sistema de arquivos se encarrega em gravar o restante dos dados em um outro lugar da flash. A segunda estrutura se refere ao controle dos `raw data`, e são gravados no sentido do fim para o início do setor, e representado na figura 5.2 como a parte inferior do desenho. Estes são chamados de `data control` e possuem uma estrutura fixa, para que o método de montagem do volume seja

rápido e eficiente. A terceira estrutura, chamada de `sector head` ou cabeçalho, possui informações de controle do sistema como por exemplo, o número de apagamentos do setor, etc. A seguir é mostrado em detalhes cada uma destas estruturas.

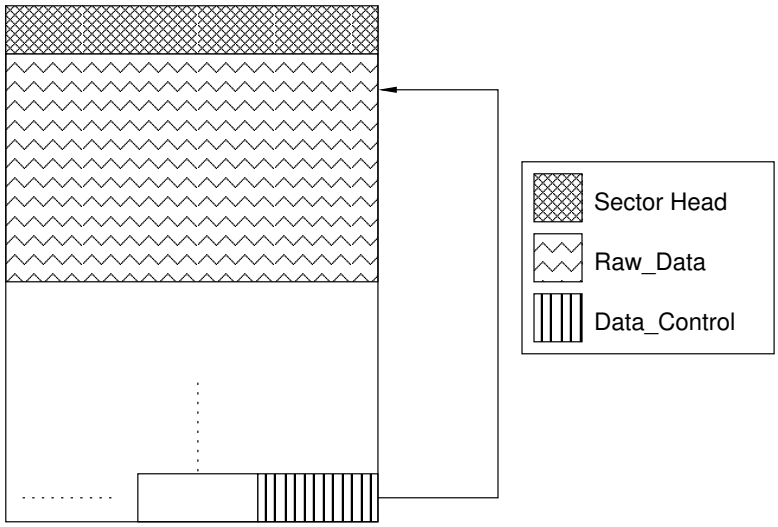

Figura 5.2: Arquitetura do Setor.

- **Área de Controle:** são estruturas usadas para melhorar o desempenho da rotina de montagem do volume. Elas estão sempre presentes no final dos setores, e são gravadas no sentido do final para o início. Estas estruturas contém referências aos `raw data` dentro do setor. Os campos desta estrutura são: `tamanho`, `offset`, `versão`, `identificador` e `tipo`. O campo `tamanho` indica a quantidade de bytes que o `raw data` ocupa. O campo `identificador` é o número lógico do arquivo, também chamado de `file id`, ao qual o `raw data` pertence. Este número é único dentro do sistema de arquivos. O campo `offset` é o local dentro do setor onde o `raw data` está armazenado. A `versão` é um número correspondente à que parte dentro do arquivo o `raw data` pertence. Este campo é necessário, para distinguir as diversas partes de um arquivo, caso existam. Caso os blocos de dados do arquivo estejam fisicamente em diferentes partes da flash, estas partes terão o mesmo `file id`, mas versões diferentes. O campo `tipo` classifica o `raw data` que esta estrutura representa. Ele pode classificar o `raw data`

em três tipos: *user data*, *log context* e *context*, como pode ser visto no próximo ítem.

- **Área de Dados:** é a estrutura que contém seus dados de acordo com o campo `tipo` do `data control` (ao qual ele pertence). Os tipos podem ser: `user data`, `log context` e `context`. O `user data` é o dado propriamente dito, gravado pelas camadas de aplicativos, através da intervenção do usuário. O `log context` é usado caso aconteça alguma atualização com o dado. O `context` pode ser entendido como a união de duas estruturas clássicas presentes na maioria dos sistemas de arquivos: uma entrada de diretório e um file descriptor. Esta última estrutura possui os campos: `nome do arquivo`, chamado de `file name`, e um `identificador lógico` do ramo superior, chamado de `father id`. Este identificador do pai é o responsável pela característica da árvore reversa. Convém ressaltar que esta estrutura `raw data` corresponde ao bloco lógico de dados de tamanho variado.

- **Cabeçalho:** estrutura presente no começo de cada setor da flash, e possui como atributos um *magic number*, um `identificador` (*sector id*), e o `número de apagamentos do setor` (*erased no*). O *sector id* é um número lógico atribuído ao setor na formatação da memória, e é responsável por diferenciar os setores de um sistema de arquivos operando com mais de uma memória flash. O número de apagamentos é usado pelo coletor de lixo e pelo método de alocação, a fim de que todos setores se deteriorem por igual. Convém ressaltar que um setor é dito vazio quando este contém apenas o cabeçalho.

Estas três estruturas apresentadas acima são manipuladas de acordo com os módulos de software, mostrados no capítulo 6.

Blocos Lógicos: Foi adotado para este trabalho o conceito de blocos lógicos de tamanho variado. Pelo fato da memória flash ser acessada a nível de bytes, a implementação de blocos de tamanho variado acaba ficando mais simples. O tamanho do bloco lógico está

limitado ao tamanho do setor em que este está inserido. O nome dado ao bloco lógico de dados neste trabalho foi `raw data`.

Gerenciamento de Blocos Livres: Pelo fato das memórias flash trabalharem com o apagamento por setor, o gerenciamento de blocos livres também segue esta filosofia. É mantido na memória principal uma lista de setores vazios, sendo que o primeiro desta lista corresponde ao setor com o menor número de apagamentos. Para garantir que os setores se deteriorem por igual, esta lista é ordenada pelo número de apagamentos em forma crescente. Quando houver a necessidade da escrita de dados, é alocado a quantidade de espaço requerido dentro do primeiro setor desta lista. Se caso não houver espaço suficiente, é retornado o número de dados alocados, e então é necessário que seja requisitado mais espaço. Detalhes de implementação do gerenciamento de blocos livres pode ser encontrado na seção 6.1.3.

5.2.3 Gerenciamento de Diretórios

Arquitetura de árvore diretamente encadeada é proposta pelos sistemas de arquivos convencionais, para uma ligação das estruturas gravadas na flash. Dentro desta arquitetura, os diretórios contém informações dos arquivos e subdiretórios pertencentes a ele, caso existam. Isto ocorre em sistemas de arquivos convencionais, como o *Unix Fast File System* (UFFS) [8]. Seguindo este método de armazenamento, os diretórios são criados com um tamanho pré-definido pelo sistema. Se por um lado não existe algum arquivo pertencente ao diretório, existe um desperdício de espaço, e por outro lado, se existem vários arquivos, e a taxa de atualização é alta, existe também um desperdício de referências, só que desta vez, inválidas.

Este projeto propõe um um sistema de diretórios armazenados como uma árvore reversamente encadeada. As estruturas pertencentes a esta árvore possuem todas as informações a seu respeito, eliminando referências diretas no sistema. No entanto, a navegabilidade de uma árvore reversa não é adequada para um sistema de arquivos, e por esse motivo, é preciso construí-la na memória RAM de uma forma direta.

O diretório é implementado como um arquivo, e assim possui todos seus atributos, como: um contexto, uma lista de arquivos, e um tipo (mais detalhes sobre arquivos, ver seção 5.2.1). Pelo fato do diretório ser implementado como um tipo especial de arquivo, ele possui todos atributos de um arquivo normal. Seu contexto é idêntico ao explicado anteriormente, com os mesmos atributos. Com isso, cada diretório possui uma referência ao seu pai, e assim sucessivamente, caracterizando uma árvore reversa.

A figura 5.3 mostra três tipos de visão dos diretórios dentro deste projeto. Na figura 5.3(b) é apresentado um exemplo de como o diretório de nome `dir1` é visto quando carregado na memória RAM. A figura 5.3(a) mostra uma visão lógica da árvore reversa de contexto. É possível observar que o contexto dos arquivos `file1.txt` e `file2.txt` possuem como contexto pai o diretório `dir1`. A figura 5.3(c) mostra o conteúdo dos três contextos apresentados quando gravados na memória flash.

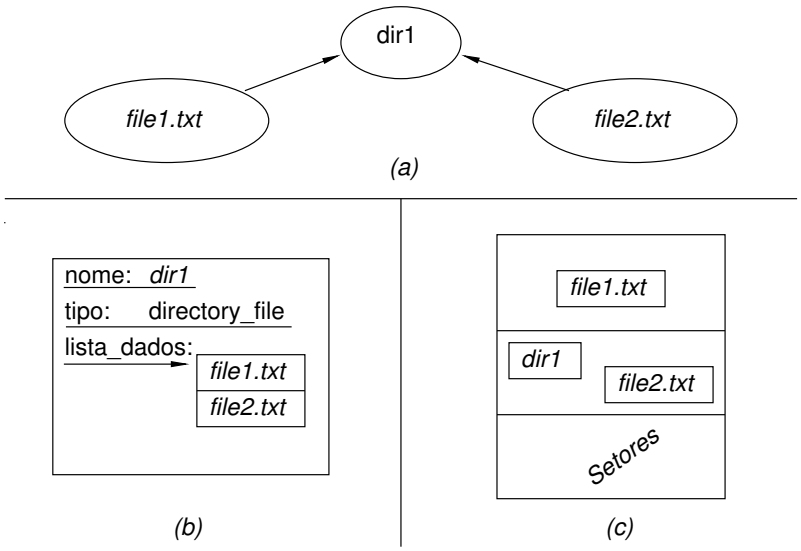

Figura 5.3: (a) Visão Lógica. (b) Visão na RAM. (c) Visão na Flash

Capítulo 6

Implementação do Sistema RIFFS

Este projeto possui sete módulos para o sistema de arquivos, conforme figura 6.1. Estes módulos são: **flash controller**, **device manager**, **allocator**, **garbage collector**, **scanner**, **file manager**, e **directory manager**, explicados a seguir.

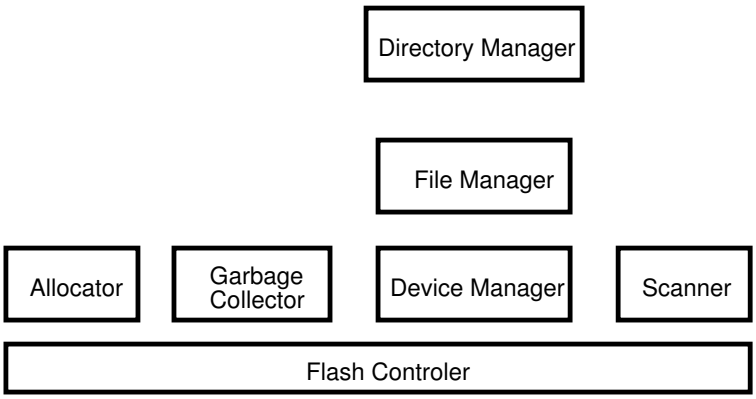

Figura 6.1: Módulos da arquitetura RIFFS.

1. **Flash Controller:** módulo responsável pela implementação das funções básicas como leitura, apagamento e escrita.
2. **Scanner:** módulo responsável pela inicialização do sistema. Percorre o final de cada setor da flash, procurando por dados válidos, armazenando-os em uma fila.
3. **Device Manager:** módulo intermediário entre árvore direta e árvore indireta. Ele é responsável pela escrita e leitura dos dados de acordo com a semântica do RIFFS.

4. **Allocator:** módulo responsável pelo gerenciamento do espaço disponível na flash. Possui uma lista de espaços vazios.

5. **Garbage Collector:** módulo responsável pela implementação das políticas de apagamento. Possui um lista de dados que precisam ser apagados.

6. **File Manager:** módulo responsável pelo gerenciamento dos arquivos propriamente dito. É subdividido em dois sub-módulos para compatibilizar as duas arquiteturas (tanto física quanto a de navegação).

7. **Directory Manager:** módulo responsável pelo gerenciamento da árvore direta. Não tem conhecimento das estruturas reversas.

6.1 Componentes do Sistema

A implementação dos sete componentes do RIFFS é mostrada a seguir.

6.1.1 Flash Control

Este módulo é responsável pela implementação das rotinas básicas de manipulação de uma memória flash. Deste jeito é possível compatibilizar todo o código de gerenciamento com os outros módulos do sistema de arquivos para várias memórias do mercado, bastando apenas implementar esta camada.

As funções implementadas por este módulo estão descritas a seguir:

- **`erase_flash`**: função responsável por apagar toda a mídia.
- **`erase_sector`**: função responsável por apagar um setor da flash.
- **`write_flash`**: responsável por escrever uma sequência de caracteres na flash.
- **`read_flash`**: função responsável por ler uma sequência de caracteres da flash.

Pelo fato de diferentes fabricantes possuírem diferentes memórias no mercado, este módulo conta com a ajuda de um sub-módulo chamado de `device geometry`.

6.1.1.1 Device Geometry

Este módulo tem o conhecimento da geometria da flash, que são: tamanho da flash, posição e tamanho dos setores. Estas informações podem ser obtidas de duas maneiras: implementadas estaticamente para cada modelo de flash, ou automaticamente através de uma interface padrão chamada de *Common Flash Interface* (CFI) [1].

Este sub-módulo implementa uma estrutura `sector` que armazena informações básicas de cada setor da flash. Estas informações são: o *tamanho do setor* e seu *endereço base*. Esta classe pode ter seus atributos armazenados tanto estaticamente (via implementação) quanto dinamicamente (via interface CFI). A interface exportada por este sub-módulo utiliza esta estrutura como retorno e parâmetros de funções. A seguir é mostrado as funções deste sub-módulo:

- **`get_sector:`** função responsável por retornar um `sector`. Possui como parâmetro o índice do setor da flash.
- **`sector_count:`** retorna o número de setores.
- **`flash_size:`** retorna o tamanho da flash.

6.1.2 Scanner

Esta camada é utilizada na montagem do volume, e depois destruída. Sua principal função é construir uma lista de `data control` que fornecerá o suporte aos demais módulos em suas inicializações. Este módulo implementa uma estrutura chamada de `data control`, que será mostrado a seguir, junto com as funções do módulo:

- **`data_control:`** é a estrutura presente ao final de todos os setores, e contém informações sobre os `raw data`. Os campos desta estrutura são os mesmos mostrados anteriormente (*size*, *id*, *offset*, *type*, *version*) no ítem 5.2.2.
- **`init:`** função que percorre o final de todos os setores, coletando os `data control`, e colocando-os em uma fila.

- **`seek:`** função que posiciona o ponteiro da fila de `data control`.
- **`next:`** função que retorna o próximo elemento da fila. Se for preciso acessar um ítem de índice especificado, é necessário chamar a função `seek` antes.
- **`count:`** função que retorna o número de `data control` coletado durante a função `init`.

6.1.3 Allocator

Camada responsável pelo gerenciamento do espaço livre na flash. Ela possui uma lista de setores limpos, ordenados pelo número de apagamentos, em uma ordem crescente. O algoritmo usado para implementação de alocação de espaço é o *first-fit*. Desta forma, caso haja uma solicitação de espaço, este módulo sempre pega o primeiro espaço livre dentro do primeiro setor da lista.

Este módulo exporta apenas uma função de alocação `alloc`, e implementa uma estrutua de controle `memory data control`. Esta classe de controle é exportada pela interface do módulo e é necessária como parâmetro para a função `alloc`. Os ítens deste módulo são explicados a seguir:

- **`memory_data_control:`** esta estrutura possui esse nome porque corresponde à estrurua `data control` (existente somente na flash), carregado na memória. Seus campos são os mesmos que o `data control` (*data offset*, *size*, *id*, *type* e *version*), e mais um campo chamado de `control offset`. Este campo é usado pelo `allocator` na função de alocação, e utilizado pelo `device manager`, para escrita dos dados.
- **`alloc:`** função responsável por implementar as políticas de alocação do sistema. Possui como parâmetro uma estrutura `memory data control`. Seu campo `size` é preenchido com a quantidade de bytes que se deseja alocar. Este método se encarrega de preencher os campos `data offset` e `control offset`, necessários para escrever um bloco na flash (mais detalhes sobre blocos lógicos do RIFFS, ver seção 5.2.2). Os outros campos desta estrutura são preenchidos por outros módulos.

Caso não seja possível alocar todo tamanho requerido, o campo `size` possuirá a quantidade que pode ser alocada. Caso o campo `size` seja igual a zero, não existe possibilidade de alocação de espaço no dispositivo.

6.1.4 Device Manager

Este módulo é responsável por implementar a escrita dos dados de acordo com a semântica do RIFFS. Esta classe também pode ser entendida como a conexão entre a arquitetura das estruturas físicas (presente na flash) e a arquitetura das estruturas lógicas (presente na RAM). Este módulo exporta duas funções, uma de escrita e outra de leitura, mostradas abaixo:

- **`write_data:`** Esta função possui dois parâmetros: `buffer` e `memory data control`. O `buffer` contém os dados que se deseja escrever na flash, e o `memory data control` contém informações básicas a respeito do `buffer`.
- **`read_data:`** Esta função possui os mesmos parâmetros da `write_data`, mas ela realiza a operação de leitura da flash. Os dados lidos são copiados para o `buffer` da memória passado como parâmetro.

6.1.5 File Manager

Módulo responsável por agrupar as diversas partes dos arquivos (caso existam) em estruturas do tipo `file`. Após o seu início, uma lista de arquivos é construída, que servirá como suporte para o início do módulo `directory manager`, explicado a seguir.

A estrutura `file` é definida por este módulo e possui os seguintes campos: `type`, `id`, `context` e `list`. O campo `type` possui o tipo do arquivo e classificação em `user file` e `system file` (também chamado de diretório). O campo `id` é o identificador lógico do arquivo. O campo `context` contém informações de controle do arquivo e, por último, o campo `list` é um ponteiro para uma lista que deve ser utilizada de acordo com o campo `tipo`. Caso o tipo seja `user file`, a lista contém referências

para estruturas do tipo `raw data` e também é chamada de `data list`. Caso o tipo seja `diretório`, a lista contém referências para classes do tipo `file`, e neste caso é chamada de `file list` (explicado em 6.1.6).

As funções implementadas por este módulo, são mostradas abaixo:

- **`init:`** função executada apenas na montagem do volume. Percorre toda a lista de `data control` do `scanner`, agrupando os blocos dos arquivos. Os blocos do tipo `raw data` que contém o mesmo identificador, são acrescentados na lista `data list` do arquivo correspondente. Este procedimento é o mesmo para os blocos do tipo `context`, com exceção que eles são adicionados em um campo especial do arquivo: `file context`. Depois de ter percorrido toda a lista do `scanner`, é feito uma verificação dos arquivos inválidos: arquivos marcados como apagados, ou os arquivos que não possuem um bloco `file context` associado, são removidos.

- **`create_file:`** Esta função cria a estrutura `file` na memória e associa um identificador ao arquivo. Os atributos `data list` e `context` são iniciados com zero e o campo `type` é iniciado com `user file`. O retorno da função é a posição de memória que a estrutura foi criada. Se o retorno da função for zero, significa que ocorreu algum erro.

- **`remove_file:`** Função responsável por gravar os atributos do arquivo como removido. O campo `versão` do bloco de dados `context` é gravado com o valor zero. Isto indica que um arquivo está apagado, pois não existe contexto com versão zero. Feito isso, uma função do módulo `garbage collector` é chamada, pois este é o módulo responsável pelo apagamento das estruturas.

- **`open_file:`** Função que abre um arquivo, e aloca recursos no sistema de arquivos. Possui como parâmetro o `id` do arquivo que se deseja abrir. O retorno da função é um ponteiro para a memório onde a estrutura `file` foi criada.

- **`close_file:`** Função que libera os recursos alocados pela abertura do arquivo.

- **`seek:`** Função responsável pelo posicionamento do ponteiro de dados do arquivo.

- **`read:`** Função que lê uma quantidade de bytes de um arquivo, e armazena em uma posição de memória. O número de bytes e a posição de memória são passados como parâmetro.

- **`append:`** Insere uma quantidade de bytes no final de um arquivo que estão em um endereço de memória.

- **`truncate:`** Função que retira os bytes que estão referenciados pelo intervalo com início no ponteiro de dados (posicionado pela função `seek`) até o fim do arquivo.

6.1.6 Directory manager

Módulo responsável pela construção e manipulação de uma árvore direta na memória principal do sistema. O `diretório` é implementado como um arquivo do tipo `system file`. O primeiro diretório é chamado de `raiz` ou `root` e seu nome é o caracter `/`. Este diretório não está gravado na flash. Ele possui um identificador padrão de número 1. Desta forma, todos arquivos que contém o campo `father id` igual à 1 pertencem ao diretório `raiz`, e então são acrescentados à lista de arquivos do diretório. O mesmo ocorre com os subdiretórios, e assim sucessivamente. As funções implementadas por este módulo são descritas a seguir:

- **`init:`** Esta função é executada somente no início do sistema. A lista de arquivos do módulo `file manager` é percorrida, e então uma árvore direta é construída na RAM.

- **`create_dir:`** Esta função é similar a função `create_file`, mas com a exceção de que a estrutura de retorno possui o campo `type` com o valor `system file`.

- **`remove_dir:`** Esta função remove um diretório vazio do sistema. Sua operação é idêntica a função `remove file`.

- **`link`:** Esta função é responsável por inserir um arquivo ao diretório. O arquivo é associado ao diretório através de um nome, que é passado como parâmetro para a função. Convém lembrar que cada operação de `link` sobre um arquivo sobrescreve o seu contexto anterior. Desta maneira, é implementado de uma maneira transparente as funções de renomear um arquivo e de mover um arquivo para outro diretório.

- **`unlink`:** Esta remove a referência `father id` do contexto do arquivo, e remove a referência do arquivo da lista de arquivos.

- **`lookup`:** Esta função procura por um arquivo cujo nome é passado como parâmetro, dentro da lista de arquivos de um diretório. O retorno da função é uma referência ao arquivo, ou caso o arquivo não exista, é retornado um erro.

6.1.7 Garbage Collector

Pela característica da flash, quando um dado precisa ser atualizado, ele é invalidado e reescrito em outro lugar. Assim, dentro do dispositivo podem existir resíduos de dados atualizados, que precisam ser apagados posteriormente. Este módulo é o responsável pelo gerenciamento dos dados inválidos para o sistema. O `garbage collector` possui duas listas: `erased list` e `invalid list`. A primeira lista é formada por objetos do tipo `sector Header`, e ordenada de forma crescente pelo campo `erased times`. A segunda lista é formada por estruturas do tipo `invalid data statistic` e ordenada de forma decrescente pelo campo `invalid data size`. A lista `erased list` é usada quando se deseja descobrir qual o destino dos dados válidos de um setor que está sendo reciclado. A lista `invalid list` é usada para selecionar o setor que possui a maior quantidade de dados inválidos para então ser reciclado. O primeiro elemento da lista `invalid list` é o setor que será escolhido para ser reciclado, quando o método `execute` for invocado. De acordo com a literatura o método de escolha de setores `origem` e `destino` usado neste trabalho é chamado de `greedy`. A seguir é mostrado as funções e classes implementadas por este módulo.

- **`invalid data statistic`:** Esta estrutura é responsável por armazenar uma estatística da quantidade de bytes inválidos de cada setor. Ela possui dois campos: `invalid data size` e `sector id`. O primeiro campo possui o total de bytes inválidos referente ao setor identificado pelo segundo campo.

- **`init`:** Esta função é executada somente no início do sistema. Ela percorre toda a lista do módulo `scanner` para detectar os dados inválidos, e começar suas estatísticas.

- **`remove_data_control`:** Esta função é usada quando se deseja retirar um bloco de dados aleatório do sistema. O bloco não é removido fisicamente no mesmo instante da invocação da função, mas sim quando for realizado a reciclagem do setor. É passado como parâmetro para esta função um objeto do tipo `data control`, e através dos campos deste objeto, é possível coletar informações suficientes para removê-lo. Um bloco de dados físico na flash é dito apagado quando a estrutura `data control` presente no final do setor, possuir seu campo `id` igual a zero. Desta maneira é possível diferenciar um bloco de dados válido de outro inválido.

- **`remove_file`:** Função responsável por retirar um arquivo do sistema. Possui como parâmetros o arquivo que se deseja remover. O arquivo em questão não é removido fisicamente no mesmo instante da chamada da função, mas sim na reciclagem do setor em que estão seus blocos de dados. A implementação desta função apaga cada um dos blocos de dados pertencentes ao arquivo através da função `remove_data_control`.

- **`execute`:** Função que inicia a execução do `garbage collector`. Ela seleciona o primeiro setor da lista `invalid list` (seleção do setor de origem), e o primeiro setor da lista `erased times` (seleção do setor de destino), e inicia a cópia dos dados válidos. A cópia é feita a partir do setor de origem para o setor de destino. Para cada bloco de dados copiado de um setor para outro, é preciso atualizar a sua referência, dentro do arquivo na memória RAM. Este conceito foi visto no gerenciamento de dados em flash, chamado de remapeamento.

6.1.8 Notas

O compromisso deste trabalho é fazer o armazenamento de árvore reversa de diretórios, sem comprometer a navegabilidade de um sistema de arquivos. Como pontos `positivos` podemos citar a baixa fragmentação causada por este projeto, pois na flash é escrito somente o necessário. Como pontos negativos este primeiro projeto não suporta `hard links`. Outra desvantagem encontrada foi no apagamento de arquivos. Nele é preciso percorrer todos os dados da lista de blocos de um arquivo, invalidando-os.

6.2 Resultados do Sistema

Após a construção do primeiro protótipo, foram realizados alguns testes práticos comparando o sistema RIFFS e o JFFS2. Entre estes testes, a comparação entre a banda de escrita dos dois sistemas provê o resultado mais interessante. Esta seção mostra alguns dados do sistema RIFFS, e algumas comparações com o sistema JFFS2.

6.2.1 Plataforma de testes

Para cada sistema de arquivos foi usado uma plataforma diferente. Esta solução foi usada pelo motivo de não conseguir uma versão do JFFS2 para a mesma plataforma RIFFS e vice-versa. Ainda por este motivo foi preciso achar algum meio de compatibilizar os testes de dois sistemas em plataformas diferentes. O meio encontrado foi achar uma **porcentagem** ou uma **proporção** do tempo de escrita[1] de dados em uma flash (através de simples funções fornecidas pelo fabricante), pelo tempo de escrita de dados através da camada do sistema de arquivos. Com isso obtém-se uma proporção da defasagem de tempo de escrita em arquivo em relação a simples funções de escrita na flash. A seguir é mostrado em detalhes as plataformas de testes e os cálculos realizados.

[1] A princípio, foi feito o teste de escrita de dados, mas este teste poderia ser de tempo de apagamento, retardo de apagamento, tempo de montagem do volume, espaço de memória ocupado em RAM, etc.

Plataforma RIFFS: A plataforma de testes utilizada neste projeto foi desenvolvida por uma empresa que atua na área de sistemas embutidos para telecomunicações, Khomp Solutions. O nome da plataforma é KMB01, e será usada futuramente como uma plataforma de convergência entre redes comutadas por circuitos e redes comutadas por pacotes. A KMB01 possui um processador PowerPC 405GP de 200MHz, da IBM. No barramento do processador estão conectados: a memória flash, a memória RAM, além de outros componentes não citados por não fazerem parte do escopo deste trabalho. A memória flash possui uma conexão com o processador através de um barramento de 16 bits e seu tamanho é igual a 2MB. O tamanho da memória RAM no sistema é de 16MB. Atualmente não existe um sistema operacional para esta plataforma, e por este motivo os testes do sistema RIFFS realizados foram funções simples. A comunicação com a plataforma de testes é feita via interface serial.

No início do projeto, houve uma dificuldade no que diz respeito á depuração, compilação e testes em geral. A empresa fabricante da placa, disponibilizava um monitor, gravado em uma EEPROM, que fazia o *boot* inicial da placa em geral (processador e periféricos). Este monitor aceitava o carregamento de um programa binário compilado para o processador da placa, obviamente, em um formato específico `srec`[2]. Ao início a depuração era feita através dos `LEDs` da placa, pelo motivo de não possuir outro meio de depuração. Aos poucos foram desenvolvidas ferramentas de depuração via interface serial, com o andamento do projeto. O sistema não possuía um sistema operacional, e assim sendo os programas de teste rodavam no modo `super usuário` do processador. Ao término dos testes, a plataforma era necessariamente religada.

Plataforma JFFS2: A plataforma utilizada para fazer os testes do sistema JFFS2 foi um iPaq H3600 da Compaq. O H3600 possui um processador StrongArm SA-1110 de 206MHz, da Intel. No barramento do processador estão conectadas: a memória RAM e a memória flash. A memória flash possui uma conexão com o processador através de um barramento de 32 bits, e seu tamanho é 16MB. O tamanho da memória RAM do sistema é de 32MB. Os testes realizados no JFFS2 rodaram sobre o sistema operacional

[2]O formato `srec` foi desenvolvido pela Motorola e conhecido como *S record format*.

Linux, e por este motivo, sabe-se a princípio que existe a presença de um escalonador e de alguns processos do *kernel* que não podem ser desativados. Desta maneira os testes não refletirão com **exatidão**, a comparação dos dois sistemas, mas mesmo assim, servirá como base para uma futura análise.

6.2.2 Escrita de Dados

A realização destes testes baseia-se na porcentagem de desempenho perdida do sistema em relação à simples operações da flash. Este esquema possui uma fórmula bem simples explicada da seguinte maneira: suponha o tempo de escrita na flash realizado por simples funções, sem intervenção de módulos do sistema, chamado de TES, e suponha o tempo de escrita da mesma quantidade de bytes através da camada de um sistema de arquivos, chamado de TEA. Subtraindo o Tempo Escrita em Arquivo (TEA) pelo Tempo de Escrita Simples (TES) e dividindo este resultado por TES obtém-se a porcentagem do consumo de Processamento Adicional (PA) imposto pelo sistema. A fórmula é mostrada a seguir.

$$PA = \frac{(TEA - TES)}{TES}$$

Esta porcentagem de desempenho foi necessária porque os testes realizados foram feitos em duas plataformas diferentes. Para cada etapa do teste realizado ambas as plataformas foram religadas e todos arquivos foram apagados. Sabe-se também que os testes não sofreram influência do `coletor de lixo` porque ele não entra em funcionamento quando há espaço suficiente para acomodar o novo arquivo. Os resultados dos tempos são descritos a seguir.

Tempo dos testes RIFFS: O Tempo de Escrita Simples (TES) para a flash da plataforma KMB01 foi adquirido da seguinte maneira:

- Escrever cem vezes, buffers de tamanho 1MB, e calcular seus respectivos tempos. Para cada buffer escrito, é calculado o tempo de escrita, o buffer é apagado, e a plataforma é reiniciada.

- Como a média dos tempos, foi encontrado: TES = 7,307456 seg. (para buffers de 1MB).

- Como os testes são menores que 1MB, então o resultado é dividido por 1024, para achar o tempo médio gasto para escrita de buffers de 1kB. Logo: o TES = 7,13619 mseg. (para buffers de 1kB).

O Tempo de Escrita em Arquivo (TEA) para o sistema RIFFS, possui seis valores. Cada valor é calculado de acordo com os testes realizados com diferentes tamanhos de buffers para escrita. Os tamanhos escolhidos foram: 16KB, 32KB, 64KB, 128KB, 256KB e 512KB. O tempo para cada tamanho de buffer foi conseguido da seguinte maneira:

- Realizar cem vezes, um `append` do tamanho do buffer em questão. Após cada `append`, o arquivo em questão é removido, e a plataforma reiniciada.

- Calcular a média do tempo gasto nestes testes.

Por conveniência, foi colocado a coluna TES na mesma tabela. Os tempos calculados para cada buffer, é mostrado na tabela 6.2.2, a seguir.

Tamanho do Buffer	**TES RIFFS (mseg)**	**TEA RIFFS (mseg)**
16KB	114,18	114,71
32KB	228,36	236,73
64KB	456,72	501,06
128KB	913,43	965,53
256KB	1826,86	1930,52
512KB	3653,73	3739,66

Tabela 6.1: Tempo de Escrita em Arquivo (TEA) para o RIFFS

Tempo dos testes JFFS2: O Tempo de Escrita Simples (TES) para a flash da plataforma iPAQ H3600 foi adquirido da seguinte maneira:

- Escrita de um buffer de tamanho 12MB, e calcular seu respectivo tempo[3].
- Com o tempo, tempos: TES = 53,557377. (para buffer de 12MB).
- Dividindo o tempo por 12288 (12KB), encontra-se: TES = 4,43989 mseg. (para buffers de 1KB).

O TEA para o JFFS2 foi conseguido da mesma maneira do TEA para o RIFFS (mostrado acima), com a diferença que ele foi executado na plataforma iPAQ H3600. Os valores deste teste não aparecem com duas casas decimais, porque o sistema operacional não provê os últimos dois dígitos. A tabela 6.2.2 mostra os valores.

Tamanho do Buffer	TEA JFFS2 (mseg)	TEA JFFS2 (mseg)
16KB	71,04	120
32KB	142,08	220
64KB	248,15	450
128KB	568,31	860
256KB	1136,61	1600
512KB	2273,22	2900

Tabela 6.2: Tempo de Escrita em Arquivo (TEA) para o JFFS2.

Comparação dos tempos: Após realizados os testes para aquisição dos tempos, utilizou-se a fórmula do Processamento Adicional (PA) explicada anteriormente, mostrado na tabela 6.2.2, a seguir.

[3] Antes de escrever o buffer, a plataforma foi reiniciada e sabia-se antecipadamente que existia disponível na flash um espaço contíguo maior que 12MB.

Tamanho do Buffer	PA RIFFS	PA JFFS2
16KB	0,47%	68,92%
32KB	3,67%	54,85%
64KB	9,71%	58,37%
128KB	5,70%	51,33%
256KB	5,67%	40,77%
512KB	2,35%	27,57%

Tabela 6.3: Comparação de desempenho entre: RIFFS e JFFS2.

6.2.3 Codificação do sistema

Atualmente, o protótipo do sistema foi construído como uma biblioteca estática, compilado inicialmente para a plataforma de testes. Esta biblioteca não utiliza nenhuma outra biblioteca externa, além daquela utilizada pelo compilador, nem chamadas ao sistema operacional, podendo ser incorporada em qualquer outro projeto ou sistema, sem maiores problemas de dependências.

6.2.4 Tamanho de estruturas

Abaixo é descrito o tamanho de algumas estruturas deste sistema:

- Tamanho de informações de controle: As informações de controle (IC) neste projeto são: o cabeçalho de cada setor (*sector header* - SH) e a estrutura que controla cada bloco de dados (*data control* - DC). O SH está presente no começo de cada setor e possui o tamanho de 64 bytes. O DC está presente no final de cada setor e cada unidade corresponde a um bloco gravado no setor ao qual ele ocupa, e possui o tamanho de 128 bytes.

- Tamanho de arquivo: O tamanho de um arquivo dentro deste projeto é dado pelo somatório do tamanho de cada bloco de dados pertencente ao arquivo. O campo `version` da estrutura de controle `data control` define cada bloco distinto de

dados de um arquivo. Este campo reserva 15 bits como endereço do bloco, e assim sendo o Número Máximo de Blocos (NMB) de um arquivo pode ser:

$$NMB = 2^{15}.$$

$$NMB = 32k. \quad (1)$$

O tamanho máximo de um bloco é limitado fisicamente pelo tamanho de um setor, ou seja, um bloco de dados não pode ser maior que um setor. Para fazer a conta do tamanho máximo de um arquivo, tomemos como exemplo uma memória flash com todos setores iguais a 64kB e tamanho suficiente para acomodar todo arquivo. Neste exemplo o Tamanho Máximo de um Bloco (TMB) pode chegar à 64kB menos Informações de Controle (IC). Logo:

$$TMB = 64kB - IC. \quad (2)$$

Multiplicando o tamanho máximo de um bloco, pelo Número Máximo de Blocos de um arquivo, obtemos o Tamanho Máximo de um Arquivo (TMA) suportado por este projeto. Assim:

$$TMA = NMB * TMB. \quad (3)$$

Substituindo (1) e (2) em (3), temos:

$$TMA = 32k * (64kB - IC). \quad (4)$$

Sendo as informações de controle um `sector header` e apenas um `data control` por setor, temos:

$$IC = SH + DC.$$

$$IC = 64 + 128.$$

$$IC = 192. \quad (5)$$

Substituindo (5) em (4), obtém-se:

$$TMA = 32k * (64kB - 192).$$

$$TMA = 2GB - 6MB.$$

$$TMA = 1,94GB.$$

- Número de arquivos: O número de arquivos do sistema é dado pela contagem de todos os arquivos. O campo `id` da estrutura de controle `data control` define um arquivo distinto dentro do sistema. Neste campo são reservados 32 bits para o número lógico de cada arquivo, com exceção do número lógico zero que possui uma finalidade de controle. Assim sendo, o Número Máximo de Arquivos (NMA) que este projeto possui é:

$$NMA = 2^{32} - 1.$$

$$NMA = 4G - 1.$$

Capítulo 7

Conclusão

Este trabalho apresentou as características do projeto RIFFS (*Reverse Indirect Flash File System*), um sistema de arquivos para memórias flash, focado nas características destas memórias e nas estruturas de controle do software. A maior parte dos esforços se concentraram em um modo de simples estruturas escritas nas flashs.

Existem atualmente diferentes implementações para memórias flash no mercado. Algumas implementam apenas a camada de acesso ao dispositivo, como é o caso do FTL, enquanto que outros desenvolvem o sistema por inteiro, como o JFFS2, por exemplo. O primeiro tipo de sistema provê uma interface semelhante à dos discos, para compatibilização dos sistemas existentes, que não levam em conta características específicas da flash. O outro tipo de sistema é construído por inteiro e considerando as limitações das memórias. Este último tipo mistura o armazenamento de suas estruturas físicas com a navegabilidade do sistema de arquivos. Com este projeto foi possível uma validação do conceito de uma estrutura de árvore reversa para armazenamento de dados diferente dos modos convencionais.

Este projeto conseguiu provar que as estruturas escritas em memórias de difícil atualização, não precisam ser complexas, e também não precisam conter todas informações pertinentes à operação do sistema. Com um gerenciamento de blocos de tamanho variado no gerenciamento de arquivos, e árvores reversamente encadeadas no gerenciamento de diretórios, conseguiu-se um sistema de arquivos com praticamente as

mesmas funcionalidades dos demais.

Atualmente existe o primeiro protótipo do sistema construído em forma de uma biblioteca de funções. Pelo fato desta biblioteca não possuir dependências externas, este projeto é facilmente portável para outras plataformas (dependendo apenas do compilador). A desvantagem deste projeto, em relação aos outros sistemas de arquivos para memórias flash, é a falta de suporte para *hard-links* no gerenciamento de arquivos.

Os resultados obtidos até o momento são muito satisfatórios. Os testes sobre o `garbage collector` não puderam ser feitos na prática, por não ter tempo suficiente para isolar o coletor de lixo do JFFS2, mas na teoria ele funciona mais rápido, devido a simplicidade das operações. Nos testes de escrita de arquivos pequenos (aprox. 16KB), o RIFFS conseguiu um desempenho muito superior ao sistema JFFS2, também devido à simplicidade de suas operações.

Como trabalho futuro, existe a necessidade de integração com um sistema operacional. Isto já está acontecendo, dentro do Laboratório de Integração de Software e Hardware (LISHA) através do projeto EPOS [4] (Embedded Paralell Operating System). Desta maneira o RIFFS possuirá todas características de um sistema de arquivos, e inserido dentro do contexto de sistema operacional. Outra proposta para desenvolvimento futuro é agregar mais funcionalidades no sistema como controle de usuários, compactação, etc.

Referências Bibliográficas

[1] AMD. Common flash memory interface specification. Technical report, Intel Corporation and Advanced Micro Devices and Fujitsu Limited and Sharp Corporation, 1. AMD Place, Sunnyvale, CA, USA, 2001.

[2] Mei-Ling Chiang, Paul C. H. Lee, and Ruei-Chuan Chang. Managing flash memory in personal communication devices. In *IEEE International Symposium on Consumer Electronics (ISCE'97)*, pages 177–182, Singapore, Dec 1997.

[3] Rômulo Silva de Oliveira, Alexandre da Silva Carissimi, and Simão Sirineo Toscani. *Sistemas Operacionais - Série Didática*. Sagra Luzzatto, first edition, 2001.

[4] Antônio Augusto Medeiros Fröhlich. *Application-Oriented Operanting Systems*. GMD Research Series, Berlin, Ge, 2001.

[5] Steve Grossman. Future trends in flash memories. In *Proceedings of the 1996 IEEE International Workshop on Memory Technology, Design and Testing (MTDT'96)*, Singapore, August 1996. IEEE.

[6] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX Technical Conference*, pages 155–164, New Orleans, LA, January 1995. USENIX Assoc.

[7] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *In Proceedings of the 27th Hawaii International Conference on Systems Science*, volume Vol. I: Architecture, pages 451–460, January 1994.

[8] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 1984.

[9] Ruei-Chuan Chang Mei-Ling Chiang, Paul C.H. Lee. Cleaning policies in mobile computers using flash memory. *The Journal of Systems and Software*, 48(3):213–231, Nov 1999.

[10] Microsoft Co. *Linear Flash Memory Devices on Microsoft Windows CE 2.1*, online edition, December 2002.

[11] Dave Poirier. *The Second Extended File System: Internal Layout*. GNU Free Documentation License and Free Software Foundation, 2001.

[12] QNX Neutrino OS. *QNX Momentics Non-Commercial Documentation Roadmap*, online edition, December 2002. [http://www.qnx.com/developer/docs/momentics_nc_docs/roadmap/].

[13] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[14] Abraham Silberschatz, James Lyle Peterson, and Peter B. Galvin. *Operating System Concepts*. ACM Press, Reading, USA, third edition, 1991.

[15] ST Microeletronics. *Background Information: Flash Memories*, online edition, February 2001. [http://us.st.com/stonline/press/news/back2002/b979m.htm].

[16] David Tennenhouse. Proactive Computing. *Communications of the ACM*, 43(5):43–50, May 2000.

[17] David Woodhouse. *JFFS: The Journalling Flash File System*. Red Hat, Inc, 1998.

[18] David Woodhouse. JFFS: The Journalling Flash File System. In *Proceedings of the Ottawa Linux Symposium 2001*, Ottawa, Canada, July 2001.

[19] David Woodhouse. *Memory Technology Device*, online edition, December 2002. [http://www.linux-mtd.infradead.org/tech/].

[20] Michael Wu and Willy Zwaenepoel. envy: a non-volatile, main memory storage system. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 86–97, San Jose, California, United State, 1994. ACM Press.